X68000

X68000
ベスト・プログラミング入門

千葉憲昭 著

技術評論社

X68000

X68000
ベスト・プログラミング入門

千葉憲昭 著

技術評論社

X68000
ベスト・プログラミング入門

千葉憲昭 著

技術評論社

はじめに

X68000が人気機種として注目され，ある地域の調査では，PC-98(日電)をしのいで人気投票No.1になったという結果が発表されています．

これは，1つに長期にわたる日電のパソコン市場独占への反動，あるいはPCに飽きたマニアたちが新鮮さを求め始めた結果とみることができます．

もう1つの見方は，8086CPUの限界を知ったマニアたちが，16ビットで最高のパフォーマンスを得るには68000しかないという結論に達したのではないかというもので，この見方は上述のそれをさらに水面下にまで掘り下げたものです．

ちょうどこのような時期に，筆者は68000CPUを使った自作パソコンの開発に取り組み，その成果と68000CPUの解説を盛り込んだ「68000システムの製作全科(上・下巻)」を技術評論社より刊行しました．

それと並行して，技術評論社編集部よりX68000のプログラミング入門のための原稿を依頼されておりましたが，幸いにも68000機のハード，ソフトにわたるノウハウを蓄積した後だけに，X68000の構造をかなりよく「透視」することができました．

X68000でのプログラミングを楽しむには，C変換できるBASICなど素材の面白さを追求するのも1つの方法であり，68000CPUの持ち味を活かしたアセンブラによって，身近かに迫力を味わうのもよい取り組み方であると思います．さらにマルチ・ウィンドウが使えるOS-9を活用して多重処理を行なえば，1台のX68000があたかも複数台に増えたかのような感覚で使えるので，楽しみは倍増されます．

本書では，X68000がもっている機能を活用してプログラミングしたい読者のために，機器構成やDOS，プログラミング・ツールなどについて解説し，プログラムの実例も掲載しています．さらに巻末では，68000ファンにとって見逃すことのできないOS-9／68000の概要についても触れています．

その中で，BASICはC変換を意図した構成とし，一方でCはとくに初心者向けに図解を取り入れるなど，イメージがつかみやすい説明になるよう配慮しました．ハイライトのアセンブラも，機械語命令のビット構成のようなわかりにくいものは省略して，大きなブロック単位での説明によって，全体の関連が把握できるような構成としました．

ページ数の制約もあり，個々の命令について長々と説明することはできませんが，反面いろいろな命令その他の関連など，マニュアルにない部分には積極的に言及したつもりです．少なくとも，Cコンパイラ(PRO-68Kセット)などの開発ルーツを購入すれば厚手のマニュアルが何冊も付いてくるので，筆者としては読者がマニュアルを参照し得る状況にあるという前提で進めました．

したがって，本書を読み進めながら，興味があればマニュアルを参照するといった利用の仕方がベストのように思われます．あるいは，マニュアルを参照しながら，全体のつながりが不明なところは本書を利用するといった使い方も考えられます．もちろん，途中に難しいところや不必要な部分があれば飛ばし読みし，後日必要になった時点で読み返すというような読み方をしてもさしつかえありません．

最後に，本書の執筆をおすすめいただいた技術評論社の加藤博氏と編集スタッフの皆様，筆者からの難問奇問に丁寧に対応していただいたシャープの高木富士雄氏に心からお礼を申し上げます．

1989年1月

千葉憲昭

本書で用いている記号について

本書においてコマンドや命令行の説明で用いている記号は，それぞれ次のような意味をもっています．

- ●〈項目〉………その項目がその位置に書かれることを示す．
- ●〔内容〕………その内容は記述を省略できる．ただし省略時に特別な意味をもつことがあるので注意が必要．
- ●｛内容｝………その内容は繰り返し記述できる．ただし，第4部のCの説明では，ブロックの始まりと終わりを示すための“{”と“}”はそのまま記述する．
- ●｜項目｜
｜項目｜………各項目のうち，任意のものが選択できることを表わす．
- ●その他………そのとおり記述する．

目次

第1部

X68000のハードウェア

第1章

68000CPUの基礎知識

1-1　68000の信号線

図1.1は，68000の信号線の概要を示したものです．

68000は，**16ビット**CPUに分類されており，**データ線**が16本(D_0〜D_{15})あります．この本数は，16ビットのデータを同時に読み書きできるための装備で，8ビット単位(D_0〜D_7, D_8〜D_{15}のいずれか)に分割した動作も可能になっています．

アドレス線の数は23本あります．したがって，全部で，

$2^{23}=8$メガ・ワード

のアドレス空間を直接操作できます．1ワードは2バイトですから，言い換えると**16メガ・バイト**となり，これだけの領域が連続している点が68000の特徴です．8086のように64キロ・バイト単位に分断されているという問題もなく，したがってセグメントという概念もありません．このことが，グラフィックや音声データのように大量にメモリに展開するデータの処理に大きな威力を発揮します．

その他の信号線は**制御線**に分類されるもので，**非同期バス制御線**のグループはデータの読み書きに使われます．また**バス調停制御線**は，DMAC(ダイレクト・メモリ・アクセス・コントローラ)などが，メモリ・アクセスするときの衝突を回避するために用意されています．**割り込み制御線**は，周辺デバイスからの割り込みを受け付ける入力線で，**プロセッサ・ステータス**は，プロセッサの動作状態(現在読み書きしているデータの種類)を示す出力線です．**6809周辺デバイス制御線**については，X68000では$\overline{VPA}$くらいしか表面に出てきません．

システム制御線の$\overline{BERROR}$は，メモリや周辺装置の応答($\overline{DTACK}$)が遅すぎる(この場合実際は応答がないことが多い)ときの打ち切りのためのもので，コンピュータの部分故障時のデッドロックを防ぐのに使われます．$\overline{RES}$はシステム立ち上げ時の起動，$\overline{HALT}$はシステムの手動停止などに用いられます．これ

第1部では，X68000のハードウェアについて，ダイジェスト的にビット単位までメスを入れます．ここで述べられていることはかなり専門的で，初心者には難解な点が多いかもしれません．実際，BASICや大半のCプログラム，またアセンブラにおいても，これほど細かな情報を利用しなければならないケースは少なく，必要のない読者は第1部を飛ばして読んでもさしつかえありません．

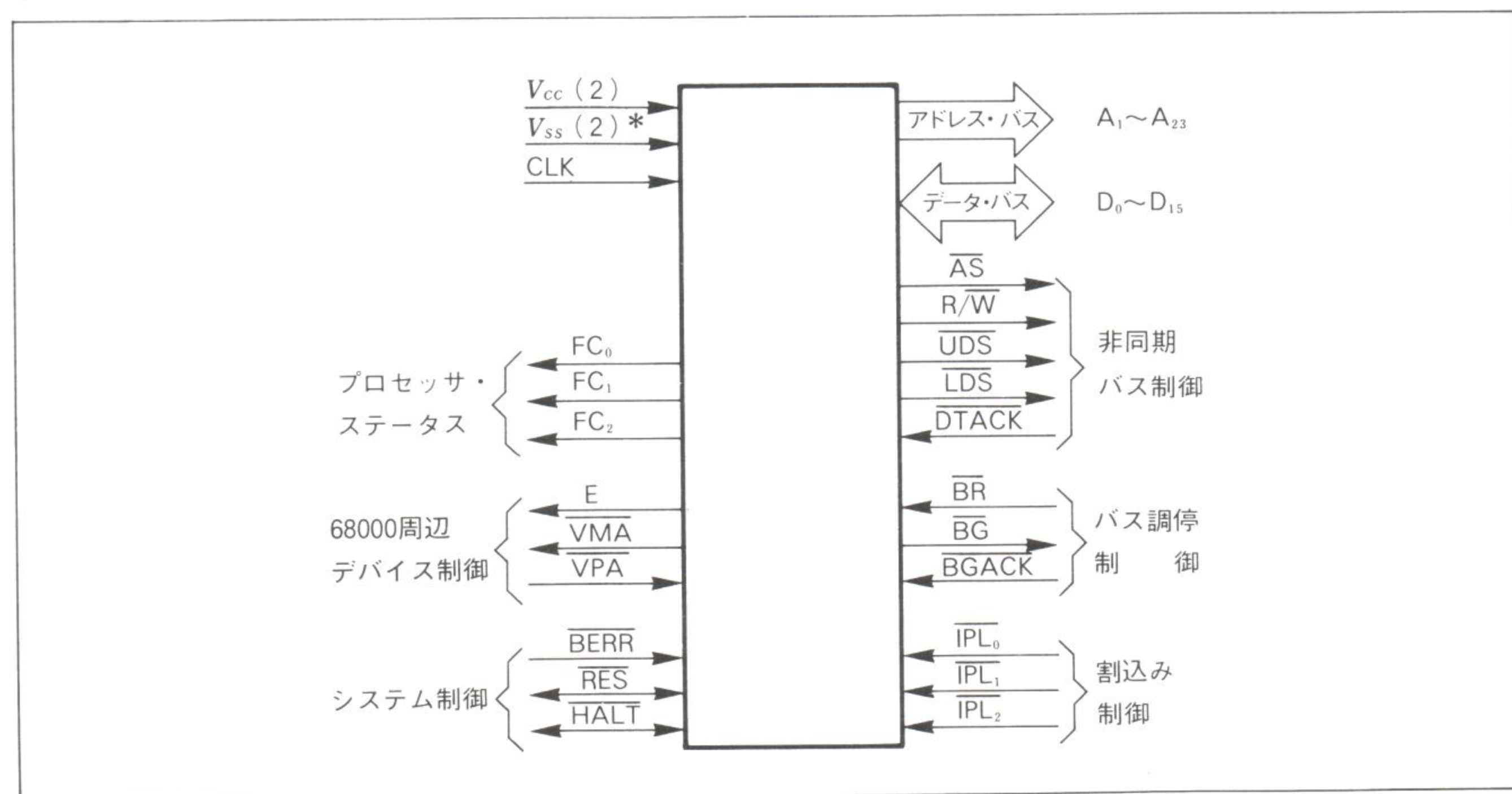
しかし，本書の構成上これらの題材は，先頭に置かなければならないもので，また，後の部でも参照する場面が出てきます．理解が進むにつれて，ここでの情報が役に立ち，高度な技術が駆使できるようになります．したがって，始めからよくわからなくても気にしないで先に進んで欲しいと思います．

第1章では，X68000以前に知っておくべき68000CPUそのものについて説明します．ただし，68000の命令については，第5部で詳述します．

らの制御線は単純ではなく，組み合わせていろいろな目的に使われるので，詳しくは拙書「68000システム製作全科(上)」などハードウェアの専門書を参照してください．

$\overline{UDS}$, $\overline{LDS}$ については，本章の1.4「メモリと周辺装置のアクセス」，FC_0～FC_2については，同じく1.6「68000

●図1．1　68000の入出力信号線

V_{CC} (2)
V_{SS} (2) *
CLK
アドレス・バス　A_1～A_{23}
データ・バス　D_0～D_{15}
$\overline{AS}$
$R/\overline{W}$
$\overline{UDS}$
$\overline{LDS}$
$\overline{DTACK}$
非同期バス制御
プロセッサ・ステータス　FC_0　FC_1　FC_2
68000周辺デバイス制御　E　$\overline{VMA}$　$\overline{VPA}$
$\overline{BR}$　$\overline{BG}$　$\overline{BGACK}$　バス調停制御
システム制御　$\overline{BERR}$　$\overline{RES}$　$\overline{HALT}$
$\overline{IPL_0}$　$\overline{IPL_1}$　$\overline{IPL_2}$　割込み制御

のプロセッサ・ステータス」，$\overline{IPL_0}$〜$\overline{IPL_2}$と$\overline{VPA}$については，同じく1.10「ハードウェア割り込みの対応」で詳述します．

1-2　68000のレジスタ構成

68000のレジスタ・セットを図1.2に示します．図を見ても明らかなように，68000は8086に比べてレジスタ数が多く，かつ汎用化されている点に特徴があります．さらに**32ビット**のデータ長をもっているため，実質的に32ビット・マシンになっているという強力さも大きなセールス・ポイントになっています．

汎用レジスタは，**データ・レジスタ**，**アドレス・レジスタ**という2つのタイプに分類されており，前者は**バイト**(8ビット)，**ワード**(16ビット)，**ロング・ワード**(32ビット)のアクセスが可能です．後者はワード・アクセスとロング・ワード・アクセスしかできず，ワード・アクセス時も値はロング・ワードに**符号拡張**されて用いられます．

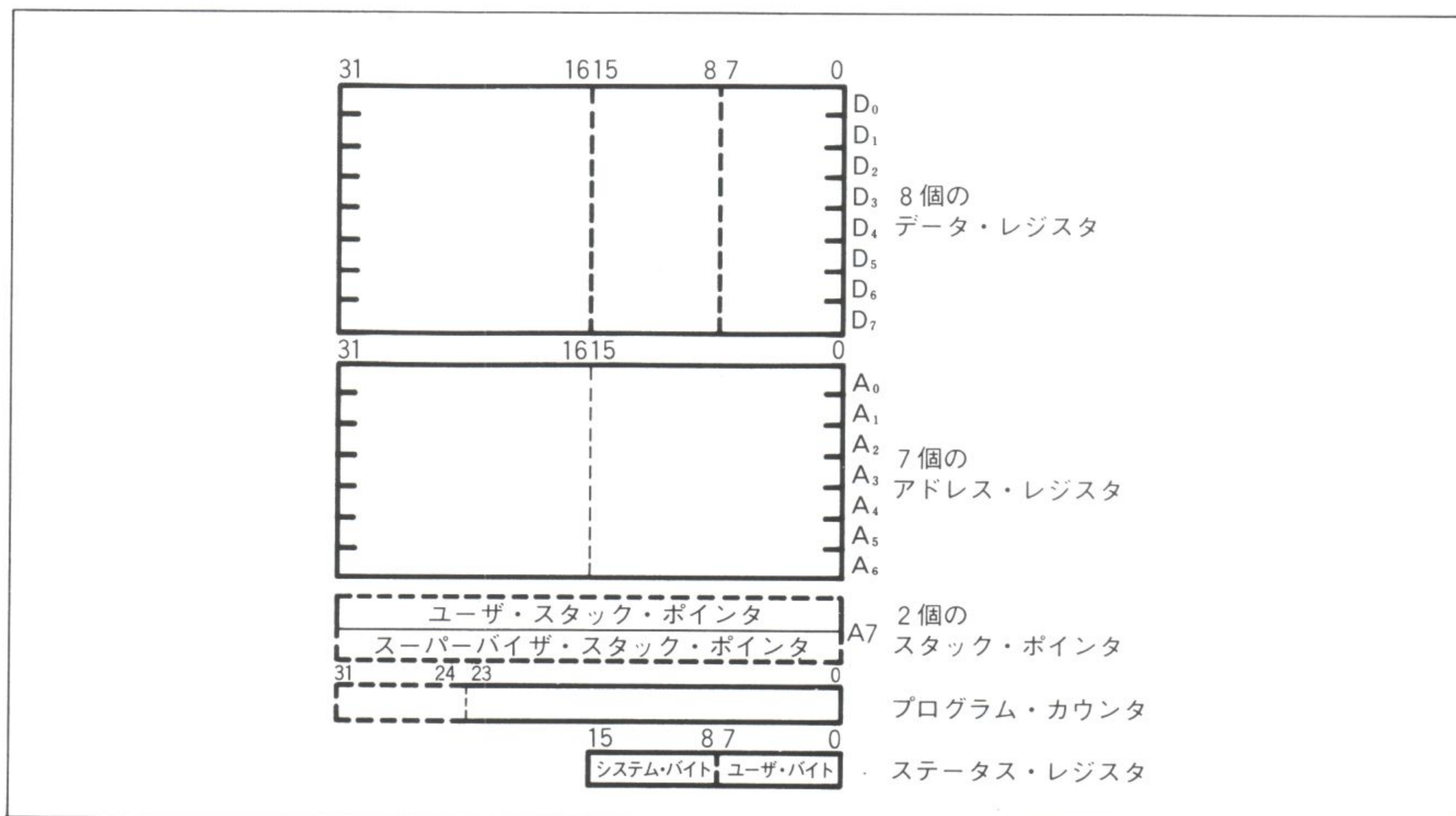
これらのレジスタは多くの命令で操作できますが，中には一方のレジスタしか操作できない命令もあります．これは，アドレス・レジスタがもっぱらアドレス値を扱うという前提によるものです．しかし，データ・レジスタ，アドレス・レジスタともに操作できる命令を使用する限りは，アドレス・レジスタさえも取り扱い方の異なるデータ・レジスタとみなしてプログラミングしてよいのです．

専用レジスタには，**スタック・ポインタ**(SP)，**プログラム・カウンタ**(PC)，**ステータス・レジスタ**(SR)の3種類が用意されています．

SPには**スーパーバイザ・スタック・ポインタ**(SSP)と**ユーザ・スタック・ポインタ**(USP)の2種類があり，ユーザ・レベルでは，USPのみが使用できます．これらの切り換えは，システムの状態により暗黙のうちに，スーパーバイザ・モードならばSSP，ユーザ・モードならばUSPが選択されることによって行なわれます．

PCは，命令の読み出しをする際のアドレス値を保持するレジスタです．この値は，ジャンプやブランチなどの専用命令でしか変更できませんが，ほかの命令でもPC相対アドレスという形で参照はできます．

SRは，唯一ワード・サイズのレジスタで，しかもユーザ・モードでは通常下位バイト(ユーザ・バイト)のみを参照します．ユーザ・バイトのことを，別名「**コンディション・コード・レジスタ**(CCR)」とも呼ん

●図1．2　68000のレジスタ・セット

でいます．

1-3 ステータス・レジスタ

ステータス・レジスタのビット構成を図1.3に示します．

このうち，上位の**システム・バイト**は，68000の場合ユーザ・モードで参照のみ可能(変更不可)です(68010，68020では参照も変更もできない)．したがって，基本的には，システム・バイトはユーザ・モードでは無視してプログラミングするのがベストです．

参考までに，システム・バイトの各ビットの意味について説明すると，次のようになります．

トレース・モードは，モニタによってトレースを実行するとき，セットされます．1のときトレースされ，1命令終了ごとにトレース処理ルーチンが起動されるため，ハードウェアの助けを借りずに，単一命令実行ができます．

スーパーバイザ状態は，1のときモニタ・プログラムが実行中，0のときユーザ・プログラムが実行中であることを示します．

割り込みマスクは，現在受け付け可能な$\overline{IRQn}$(nは1～7)線による割り込みの最低レベルから1を引いた値を表わします．レベル7を除き，この値と等しいかまたは下回る割り込みは受け付けられません．

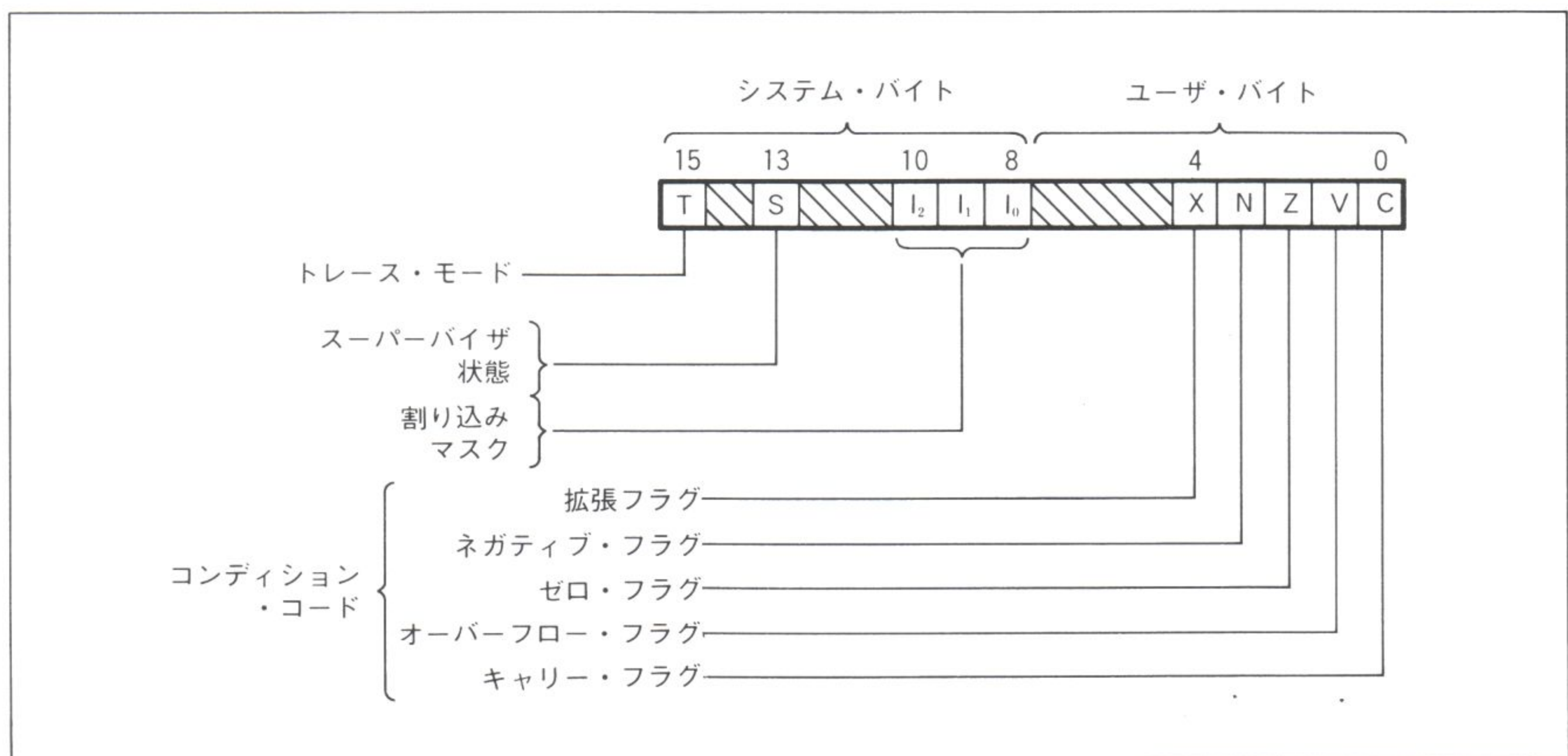
ユーザ・バイト(コンディション・コード)は，スーパーバイザ・モード，ユーザ・モードのいずれからも参照，変更が可能です．この中の個々のビットは，それぞれ演算命令などの実行によってセット，クリアされる「フラグ」です．

ネガティブ(N)・フラグは，命令の実行結果が負になったときセットされるフラグです．操作が行なわれる場合，実行結果の最上位ビットが収容されます．

ゼロ(Z)・フラグは，命令の実行結果がゼロになったときセットされます．ゼロのとき1になり，その他のとき0になる点に注意が必要です．

オーバーフロー(V)・フラグは，演算結果において，符号付き2進値としてオーバーフローが生じたときセットされます．

キャリー(C)・フラグは，演算の結果桁上がりが生じたときセットされます．この値は，命令によっては**エクステンド(E)・フラグ**にもコピーされますが，Eフラグの値を変更できる命令は限られており，この違いはたくさんの命令ステップにわたって値を保存するのに利用されています．

●図1．3　ステータス・レジスタのビット構成

1-4 メモリと周辺装置のアクセス

68000では**メモリ・マップドI/O**という考え方を採用しているため，周辺装置もメモリと同様に読み書きでき，したがって入出力専用の命令をとくに用意していません．あえて入出力を意識した命令をあげればmovepぐらいがそれに該当しますが，この命令はメモリに対しても使用可能です．このため，以下の説明ではメモリも周辺装置もとくに区別しないで扱うことにします．

さて，68000のデータ・バスのサイズは，16ビットあります．したがって1回のアクセスで読み書きできるビット数はこれが最大で，フルサイズの読み書きのことを**ワード・アクセス**といいます．レジスタのビット数は32ビットあるので，レジスタとメモリの間の全ビット転送は，ワード・アクセスを2度行なうことになり，このことを**ロング・ワード・アクセス**と呼ぶことがあります．

ワード・アクセスは，2バイト分のデータをまとめて処理するので，アドレス・ビットの最下位(A_0)はアドレス・バスに出ていません．

そこで，バイト・データ(8ビット)を転送するバイト・アクセスでは，データ・バスの上半分(D_8～D_{15})または下半分(D_0～D_7)だけを使う方法を採用しています．このため，$\overline{UDS}$(上位データ有効)，$\overline{LDS}$(下位データ有効)信号を出して，どちらが有効であるかを知らせる仕組ができています．このとき，偶数アドレスは$\overline{UDS}$側，奇数アドレスは$\overline{LDS}$側に作用します．もちろんワード・アクセスの場合は，両方が有効となります．

ワード・アクセスはデータの始まりが偶数アドレスになっているので，奇数アドレスが指定されることは考えられません．したがって，このような状態はシステムに何らかの異常が生じたとみなされ，後述のシステム・エラーとして扱われます．

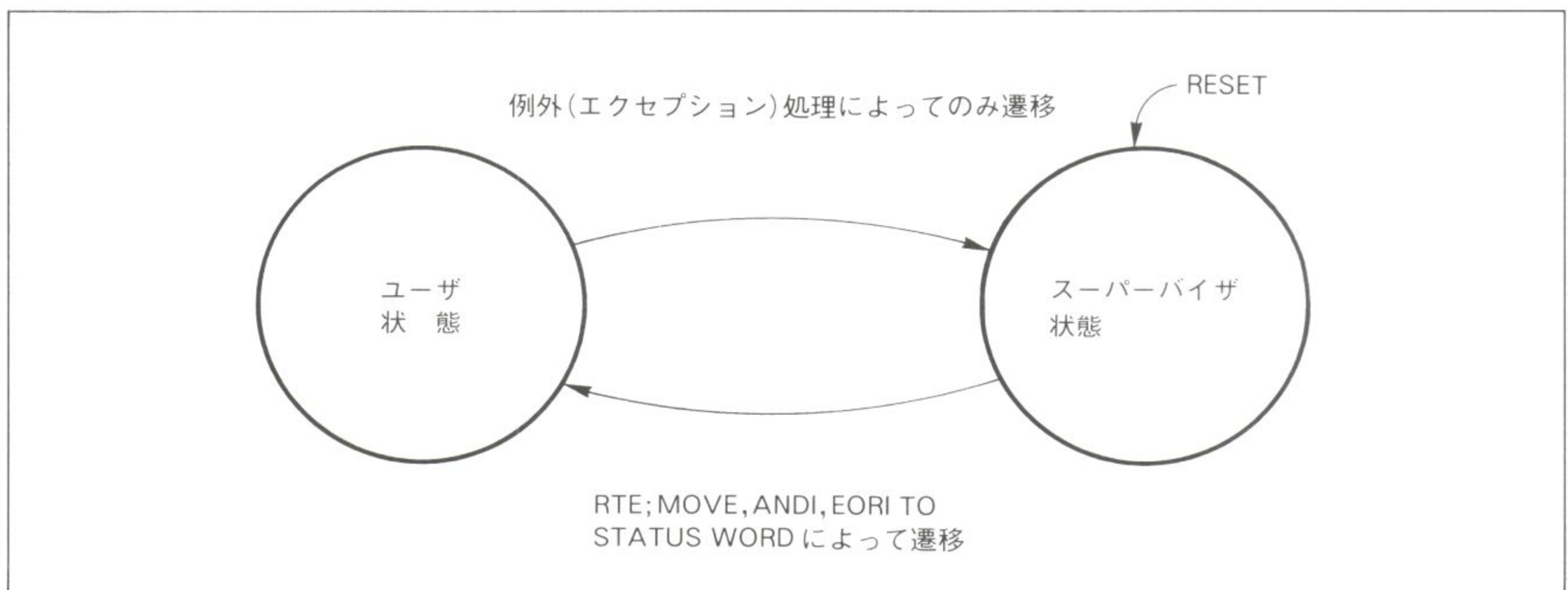
参考までに，X68000では8ビットI/Oデバイスは，奇数アドレスに配置しています．

1-5 スーパーバイザ・モードとユーザ・モード

68000には2つのモードがあります．ひとつはHuman68Kなどのモニタが動作する**スーパーバイザ・モード**，もうひとつはモニタの下で動作する一般のプログラムを扱う**ユーザ・モード**です．

これらのモードは，ステータス・レジスタの**スーパーバイザ状態**ビットによって切り換えられ，参照されます．したがって，モードを変更する方法の1つは，このビットを直接書き換えすることです．ただし，ユーザ・モードでは，ステータス・レジスタを変更する命令が使えないため，あえて行なおうとすれば，後述する特権命令違反の例外処理が起動されます．

●図1．4　68000のスーパーバイザ・モード／ユーザ・モード

例外(エクセプション)処理によってのみ遷移
RESET
ユーザ
状　態
スーパーバイザ
状態
RTE;MOVE,ANDI,EORI TO
STATUS WORDによって遷移

例外処理はスーパーバイザ・モードで行なわれるため，68000は例外発生をキャッチすると，ただちに**スーパバイザ状態**ビットに“１”を転送するようになっています．この動作は，システム起動時(リセット)においても同様に行なわれています．したがって，ユーザ・モードからスーパーバイザ・モードにするには，何らかの例外を発生すればよいので，Human68K のファンクション・コールなどではこのことを利用しています．

一般に私たちユーザは，スーパーバイザ・モードからユーザ・モードに移行するときのことを考える必要はほとんどありません．しかし例外処理ルーチンなどを作成する場合には，このための知識が必要になります．

上述のように，確かにスーパーバイザ・モードでは直接ステータス・レジスタを変更できるため，物理的には ANDI to SR などの命令によって切り換えが可能です．しかし，例外処理ルーチンの出口に使われる RTE 命令にも切り換える働きがあり，こちらのほうは切り換えと同時にユーザ・プログラムに戻るため，簡単でかつ安全確実です．

以上のことを整理すると，図1.4のようになります．

1-6　68000のプロセッサ・ステータス

68000はメモリ(周辺デバイスを含む)を参照(リードまたはライト)する際，それがどのような要求に基づいたものであるかを **FC_0〜FC_2**に出力します(表1.1)．

ユーザ／スーパーバイザの区別は，すでに説明したとおりですが，**プログラム**状態は68000がプログラムの命令をフェッチ(取得)するためのアクセスであることを表わし，**データ**状態は命令の実行のためにリードまたはライトすることを意味します．X68000ではこの区別を利用して，ユーザ・プログラムからシステム領域をアクセスできないような歯止めを設けています．

インタラプト・アクノレッジ状態は，周辺デバイスからの割り込みを受け，応答中であることを示します．さらに詳しくは，本章1.10「ハードウェア割り込みの対応」のところで説明します．

●表１．１　68000のプロセッサ・ステータス

FC線出力			参照クラス
FC_2	FC_1	FC_0	
0	0	0	未定義
0	0	1	ユーザ・データ
0	1	0	ユーザ・プログラム
0	1	1	未定義
1	0	0	未定義
1	0	1	スーパーバイザ・データ
1	1	0	スーパーバイザ・プログラム
1	1	1	インタラプト・アクノレッジ

1-7 ベクトルとシステム・リセット動作

68000CPUは，アドレス0から1,024バイトにわたって「**例外ベクトル**」と呼ばれる特別な領域を必要としています(表1.2).

たとえば，電源ONやリセットによるシステム立ち上げ時には，次に述べるベクトル内容を参照します.

● SSP初期値

立ち上げ直後のSSP(システム・スタック・ポインタ)の値を与えるもので，システム・スタック領域の末尾のアドレス＋1の値が定義されます.

● PC初期値

最初に実行する命令が収容されているメモリのアドレス値を示します.

68000CPUは，リセット信号を受けるとすべての動作を停止します．そして，解除されると同時に，ベクトル0(SSP初期値)から読み込んだ値をSSPに転送し，ベクトル1(PC初期値)の内容をPCにコピーします.

この結果プログラムは，ベクトル1が示すアドレスから実行が開始されます．このときSSPの値もすで

●表1．2　68000のシステム・ベクトル領域の内容

ベクトル番号	アドレス			内容
	10進	16進	領域	
0	0	000	SP	Reset: Initial SSP
1	4	004	SP	Reset: Initial PC
2	8	008	SD	Bus Error
3	12	00C	SD	Address Error
4	16	010	SD	Illegal Instruction
5	20	014	SD	Zero Divide
6	24	018	SD	CHK Instruction
7	28	01C	SD	TRAPV Instruction
8	32	020	SD	Privilege Violation
9	36	024	SD	Trace
10	40	028	SD	Line 1010 Emulator
11	44	02C	SD	Line 1111 Emulator
12*	48	030	SD	(Unassigned, reserved)
13*	52	034	SD	(Unassigned, reserved)
14*	56	038	SD	(Unassigned, reserved)
15*	60	03C	SD	Uninitialized Interrupt Vector
16・23*	64	040	SD	(Unassigned, reserved)
	95	05F		—
24	96	060	SD	Spurious Interrupt
25	100	064	SD	Level 1 Interrupt Autovector
26	104	068	SD	Level 2 Interrupt Autovector
27	108	06C	SD	Level 3 Interrupt Autovector
28	112	070	SD	Level 4 Interrupt Autovector
29	116	074	SD	Level 5 Interrupt Autovector
30	120	078	SD	Level 6 Interrupt Autovector
31	124	07C	SD	Level 7 Interrupt Autovector
32・47	128	080	SD	TRAP Instruction Vectors
	191	0BF		—
48・63*	192	0CO	SD	(Unassigned, reserved)
	255	0FF		—
64・255	256	100	SD	User Interrupt Vectors
	1023	3FF		—

*印は将来の拡張に備えて使用禁止

に決まっているので，最初に実行される命令がシステム・スタックを利用する命令(たとえばBSRなど)であっても困ることはありません．

なお，ベクトル０，１の参照に先立って，ステータス・レジスタの内容は，次のように初期設定されます．すなわち，**スーパーバイザ状態**の表示は１になり，スーパーバイザ・モードで実行が開始されます．**トレース・モード**は０にされ，トレースしないように働きます．**割り込みマスク**は７となり，通常の割り込みは受け付けないようにします．これによってシステム立ち上げ中の誤動作を防ぎますが，起動プログラムでは必要な時点でマスクの解除を行ないます．

1-8 システム・エラーの種類と原因

Ｃやアセンブラなどのプログラムを走らせると，「バス・エラーが発生しました」といったメッセージが表示されて中断することがあります．68000CPUは，各種のシステム上の異常状態を検出し，中断させる機能をもっており，上記の例は，そのひとつが作動した結果にすぎません．これらのエラーがどのような意味をもつかについて知っておくことは，デバック上での手掛りにもなるので，ここでは個々の原因について言及したいと思います．

●バス・エラー

読み書きしようとしたアドレスにおいて，メモリや周辺装置から一定時間内に応答がないとき，バス・エラーとなります．X68000では，ユーザ・モードでシステム領域を参照しても発生します．

●アドレス・エラー

プログラムの実行にあたり，実行開始点やジャンプ先のアドレスが奇数だったり，スタック・ポインタの値やワードまたはロング・ワード・データのアドレスが奇数のときに発生します．

●不当命令

命令のビット構成が不適当な命令(上位４ビットが1111，1010のものを除く)を実行したことを示すエラーです．

●ゼロによる除算

割り算のとき，０で割ると発生します．

● CHK命令

CHKは，データ・レジスタの値が上限値の範囲内にあるかどうかを検査する命令です．このとき範囲外だったり，マイナスだったりすると，エラーとなって中断します．

● TRAPV命令

TRAPVはオーバーフロー・フラグがセットされているとき，プログラムを中断する命令です．そうでないときこの命令は，スキップされます．

●特権命令違反

ユーザ・モードにおいて，スーパーバイザ・モードだけに許された命令を実行すると，これを排除するため実行が中断されます．

68000では，以上のもの以外についても，次節で述べるような割り込み処理を行なう機能をもっています．なお，CHK，TRAPV命令による割り込みは，必ずしもシステム・エラーとは限りませんが，ベクトルの並びの順序で説明したためこちらのほうに含めました．

これらの割り込みが発生すると，対応するベクトルの値が読み込まれます．そして，それによって示されたアドレスから臨時的な処理が開始されます．このとき，中断したポイントの次の命令から再開が可能なように，再開アドレス，ステータス・レジスタの内容がスタックに入れられます．割り込み時の臨時処理が終了すると，再開アドレスがPCに転送され，ステータス・レジスタの内容も復元されます．これによって，元のプログラムは何事もなかったかのように続行されるのです．

1-9 その他の割り込み処理

68000がもっている割り込み対応機能のうち，まだ説明していないものの働きは次のとおりです．

●トレース

ステータス・レジスタのトレース・モード・ビットが1のとき，1命令実行ごとに割り込みが発生します．デバッグのための機能です．

●1010エミュレータ

命令コードの頭4ビットが，1010のものは命令表にありませんが，不当命令とはせずにこの分類の割り込みを発生させます．割り込みルーチンでユーザが対応すれば，あたかもユーザの定義の新しい命令ができたかのように動作できます．

●1111エミュレータ

命令コードの頭4ビットが，1111の場合の割り込みです．後は1010エミュレータと同様です．

●スプリアス割り込み

ハードウェア割り込みを発生させようとして周辺装置が起動に失敗したとき，強制的に打ち切らせるために使われる割り込みです．

●自動ベクトル1～7

ハードウェア割り込みで，割り込み線1～7に対応する形での応答をするものです．

● TRAP 命令

TRAP 命令が実行されたとき，指定されたベクトル番号(0～15)に対応した割り込みが発生します．

●ユーザ割り込み

ハードウェア割り込みで,周辺装置から転送されたベクトル番号(64～255)に対応した応答をするためのものです．

●暫定割り込み

ハードウェアで周辺装置のベクトル番号が決まっていないとき,とりあえず15番を与えて動作させます．

これらの割り込みについても，前節で述べたような割り込み対応処理が行なわれます．

1-10 ハードウェア割り込みの対応

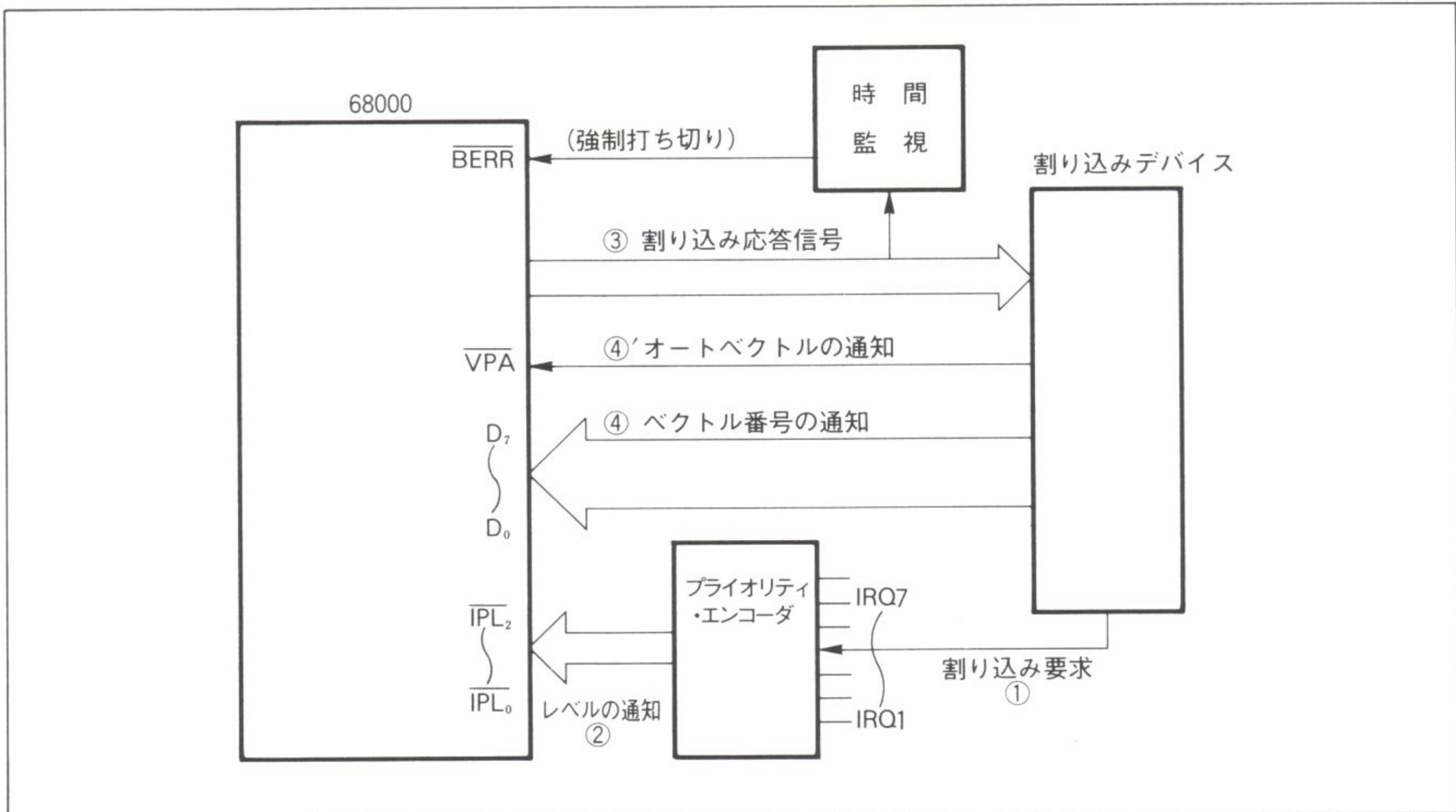
図1.5は，割り込みを発生する周辺デバイスと，68000の関係を示したものです．

割り込みを要求するときは，周辺デバイスは割り込み要求線(7本のうち1本使用)に信号(0)を出力します(①)．このとき，どの割り込み要求線を使用するかは，システム設計時に決められ，線の番号が大きいほど優先度が高くなります．

プライオリティ・エンコーダは，74LS148などを想定しており，割り込み要求を受け付けた線の番号をレベルとして，68000に通知する働きをします．もし複数の割り込み要求線に0が入力された場合は，その中で最高の番号のものが採用されます．

68000は$\overline{IPL_0 \sim _2}$で割り込み要求レベルの通知を受けると，現在処理している命令の実行が終了してから$FC_0 \sim _2$に応答信号(インタラプト・アクノレッジ)を出力します．このときアドレス線の$A_1 \sim _3$には，プライオリティ・エンコーダから受信したレベル番号を送出します．

このレベル番号は，割り込み発生源のデバイスによって参照され，要求と一致したデバイスは，以下に述べるいずれかの方法で68000に対処方法の種類を通知します．1つは，$\overline{VPA}$信号を0にするやり方で，この信号が出されると68000は**自動ベクトル**として扱い,優先度に対応するベクトルを参照して割り込みルーチンを起動します．もう1つの方法は，$\overline{VPA}$は1のままで，ベクトル番号を$D_0 \sim _7$に送出するもので，

●図1．5　ハードウェア割り込みの動作関連

68000では**ユーザ割り込み**として対応します．そして割り込みルーチンの開始アドレスは，このベクトル番号を4倍したアドレスのメモリを参照して取得します．

これらの割り込みデバイス側の④(または④′)の通知信号が，故障などで永久に出されないと，68000はそのままの状態で動くことができません．したがって，一定時間内に反応がないときは，$\overline{BERR}$に強制打ち切り信号(0)を出します．そうすると，**スプリアス割り込み**が起動され，68000のハードウェア割り込みの対応は打ち切られます．

68000による割り込みの受け付けは，常に行なわれるのではなく，SR内にある**割り込みマスク・ビット**で示される値以下の優先度の割り込みは無視されます．ただし，レベル7の場合はマスク・ビットがどのような値でも受け付けられ，「**ノンマスカブル割り込み**」と呼ばれています．

なお，割り込みのないときは，レベルは0になっています．注意が必要なのは，プライオリティ・エンコーダからのレベル信号は0と1が反転しており，たとえば3ならば011でなく100となります．したがってレベル0ならば111が68000に送られているので，68000側では$\overline{IPL_0 \sim _2}$のいずれか1本に0が入力されれば，割り込みが発生したものとして扱います．

第2章

X68000のハードウェア概要

2-1 X68000のハードウェア構成

X68000は，その名のとおりCPUに68000を搭載したマシンですが，本家の68000とは異なり，C-MOS版のHD68HC000(日立)を採用しています．このほかにもキーボード用のサブCPUとして80C51を用いたり，RTCとしてRP5C15を使用するなど，スタンバイ状態でキーボードなどからのTVコントロールが可能なように設計されています．つまり，ケース裏面の主電源スイッチを切らない限り，CPUを含めた一部の機能は働いています．

主として，機能という観点から見たX68000のハードウェア仕様は，表2.1のとおりです．また，電気信号の関連を示すブロック・ダイヤグラムは，図2.1のようになっています．

プログラムにも関連する全体のメモリ・マップ(起動後)は，図2.2に示します．ここではメイン・メモリ(RAM)が0番地から配置されており，後尾にI/O領域が張り付けられています．拡張すれば，12MBに近いユーザ領域が利用できることになります．I/O領域の中には，画像データのリフレッシュ・メモリ(VRAM)などのほかに，各種I/Oポートが配置されており，そのアドレスは表2.2のようになっています．これらの詳細については，第2章「画面制御のハードウェア」，第3章「周辺デバイスの基礎知識」などで説明します．

68000について説明したので，ここではX68000のハードウェア概要について述べます．ここで取り上げるテーマは，全体の構成と電源系統，システム起動時の動作，システム・ポートの構成などです．

システムの各部分に関する詳細については，画面関係を第3章，その他を第4章に分けて説明します．

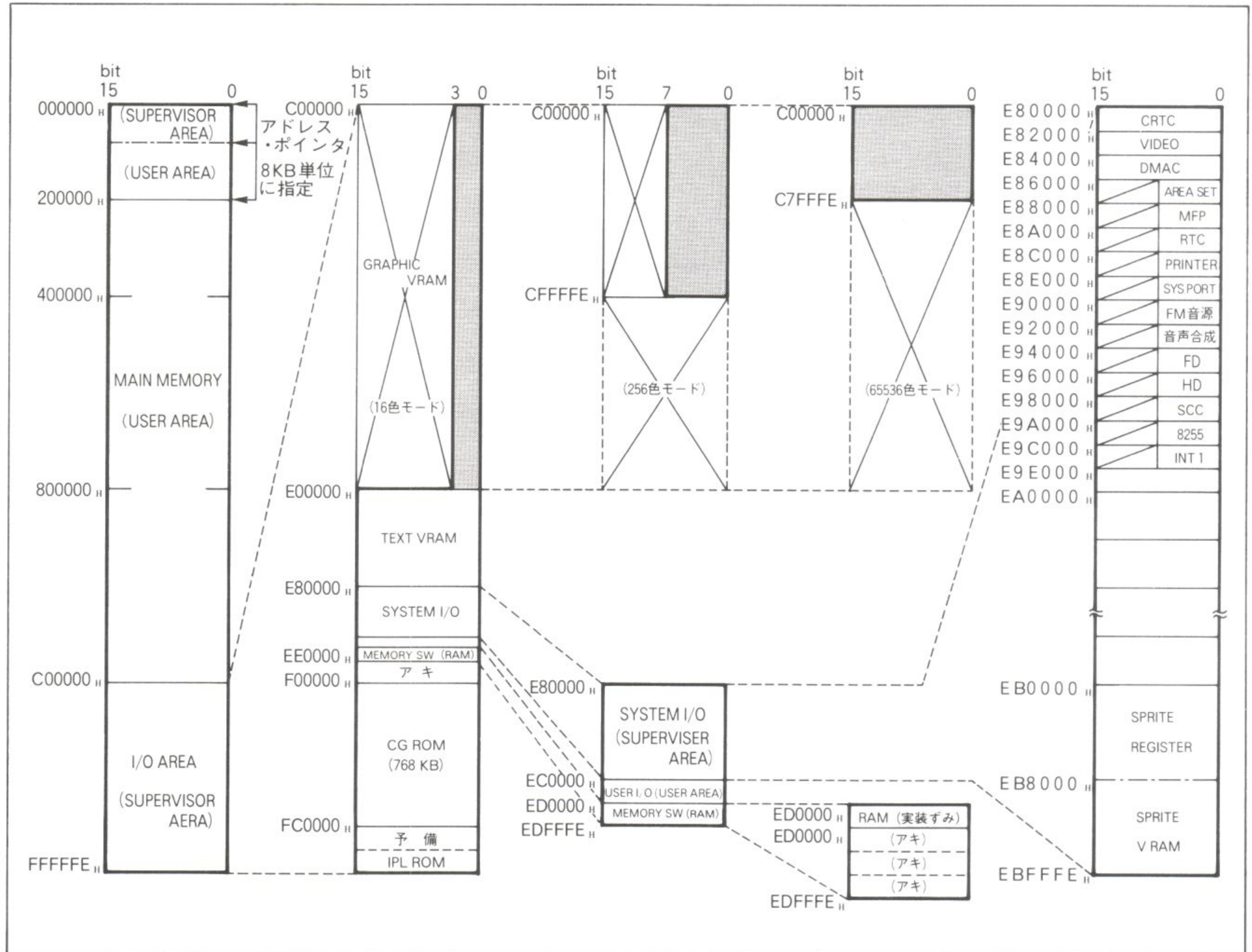
●表2．1　X68000のハードウェア仕様①

項目	分類	名称・種類	内容	備考
CPU	MPU	HD68HC000	16/32ビットMPU（10MHz）	
	サブCPU（キーボード）	MSM80C51	キーボード・スキャン	
周辺LSI	DMAC	HD63450	4チャンネルDMAC	
	MFP	MD68901	マルチ・ファンクション・ペリフェラル KEYデータの受信，各種割り込み	
	CRTC	カスタムI/C	テキスト・グラフィック制御用CRTC デュアルポートDRAMコントロール スクロール機能	スーパーインポーズのための同期合わせはCRTC内で行なう．
	スプライト・コントローラ	カスタムI/C	スプライト機能	
	FDC	μPD72065	5″2HDFDDを制御	
	ビデオ・コントローラ	カスタムI/C	パレット・プライオリティ機能，特殊モード機能	
	SCC	Z8530	シリアル・コミュニケーション・コントローラ シリアル・2チャンネル（RS-232C，マウス）	
	RTC	RP5C15	リアルタイム・クロック	
	FM音源	YM2151	8チャンネルFM音源の発音が可能	
	音声合成	MSM6258	Adaptive Differential PCM	
	PPI	μPD8255	ジョイスティック2ポート， 音声合成切り換えコントロール	
	I/Oコントローラ	カスタムI/C	フロッピーディスク，ハードディスク	
	その他	カスタムI/C	メモリ・コントローラ，システム・コントローラ	

●表2．1　X68000のハードウェア仕様②

項目	分類	名称・種類	内容	備考
メモリ	ROM	IPL ROM	256Kバイト（IPL，BIOS，メモ帳）	
		CG ROM	768Kバイト　16×16ドット，24×24ドット…全角 8×16ドット，12×24ドット…半角 8×8ドット，12×12ドット…1/4角	（JIS第一，第二水準）
	RAM	メイン・メモリ	1Mバイト	12Mバイトまで拡張可
		テキスト VRAM	512Kバイト ビット・マップ方式（設定データ横方向） 1,024×1,024ドット・4プレーン	デュアルポートDRAM採用
		グラフィック VRAM	512Kバイト ビット・マップ方式（設定データ奥行方向） 1,024×1,024ドット・4プレーン （512×512ドット・16プレーン）	デュアルポートDRAM採用
		スプライト VRAM	32Kバイト	
		SRAM	16Kバイト	
内蔵I/Fコネクタ	ディスク	内蔵ミニフロッピー	5″両面高密度（2HD）　2基内蔵	
		フロッピーディスク・インターフェース	拡張用のフロッピーディスク・ドライブ用	
		ハードディスク・インターフェース	オプションのハードディスク・ドライブ用	
	キーボード・コネクタ		専用キーボード用	
	CRTインターフェース		アナログRGB出力	
	TVコントロール・コネクタ		専用ディスプレイのTVコントロール用	
	RS-232Cインターフェース		1チャンネルRS-232C	
	マウス・インターフェース		付属のマウス用	
	プリンタ・インターフェース		セントロニクス社規格準拠	
	ジョイスティック・インターフェース		アタリ社規格準拠（2個）	
	オーディオ入出力コネクタ		ライン入出力，ヘッドホン出力	
	画像入力インターフェース		オプションの“デジタイズテロッパ”（仮称）用	
コネクタ	その他		EXPWON，VHT	
拡張用I/Oスロット			2スロット	

●図２．１　X 68000のブロック・ダイヤグラム

●図2．2　メモリ・マップ

●表2．2　I／Oポート・アドレス一覧（＊は無効）①

項　目	ポート・アドレス	リード/ライト	機　能	備　考
	E80000H	W	水平トータル	
	E80002H	W	水平同期終了位置	
	E80004H	W	水平表示開始位置	
	E80006H	W	水平表示終了位置	
	E80008H	W	垂直トータル	
	E8000AH	W	垂直同期終了位置	
	E8000CH	W	垂直表示開始位置	
	E8000EH	W	垂直表示終了位置	
	E80010H	W	外部同期水平アジャスト	
	E80012H	W	ラスター割り込み位置	
	E80014H	W	X方向スクロール	
	E80016H	W	Y方向スクロール	
	E80018H	W	スクリーン0　X	
	E8001AH	W	スクリーン0　Y	
	E8001CH	W	スクリーン1　X	
	E8001EH	W	スクリーン1　Y	
	E80020H	W	スクリーン2　X	
	E80022H	W	スクリーン2　Y	
	E80024H	W	スクリーン3　X	
	E80026H	W	スクリーン3　Y	
	E80028H	R/W	メモリ・モード，表示モードセット	
	E8002AH	R/W	テキスト・アクセス，高速クリアプレーン	
	E8002CH	W	ソース/デスティネーション・ラスター	

項目	ポート・アドレス	リード/ライト	機能	備考
CRTC	E8002EH	W	ビット・マスク・レジスタ	
	E80480H	R/W	CRTC動作設定ポート	
Gra パレット	E82000H ～ E8201EH	R/W	16色モード（256色モード または 65,536色モード）	グラフィック用パレット
	E82020H ～ E821FEH	R/W	256色モード または 65,536色モード	
Text & SP パレット	E82200H ～ E8221EH	R/W	テキスト（スプライト・カラーテーブル０）共通パレット	テキスト，スプライト用パレット
	E82220H ～ E8223EH	R/W	スプライト・カラーテーブル１パレット	スプライト用パレット
	E82240H ～ E8225EH	R/W	スプライト・カラーテーブル２パレット	スプライト用パレット
	～			
	E823E0H ～ E823FEH	R/W	スプライト・カラーテーブル15パレット	
ビデオ・コントローラ	E82400H	R/W	メモリ・モード設定	
	E82500H	R/W	プライオリティ設定	
	E82600H	R/W	特殊モード，画面表示制御	半透明，特殊プライオリティ
DMAC	E84000H	R/W	チャンネル・ステータス・レジスタ	ただし，各チャンネル・ポートのアドレスについては左記のポート・アドレス(P)にそれぞれ次のように加えたアドレスになる
	E84001H	R	チャンネル・エラー・レジスタ	
	E84004H	R/W	デバイス・コントロール・レジスタ	
	E84005H	R/W	オペレーション・コントロール・レジスタ	
	E84006H	R/W	シーケンス・コントロール・レジスタ	
	E84007H	R/W	チャンネル・コントロール・レジスタ	
	E84025H	R/W	ノーマル・インタラプト・ベクタ	チャンネル0 P+00H
	E84027H	R/W	エラー・インタラプト・ベクタ	
	E8402DH	R/W	チャンネル・プライオリティ・レジスタ	チャンネル1 P+40H
	E84029H	R/W	メモリ・ファンクション・コード	
	E84031H	R/W	デバイス・ファンクション・コード	チャンネル2 P+80H
	E84039H	R/W	ベース・ファンクション・コード	
	E8400AH	R/W	メモリ・トランスファ・カウンタ（Word）	チャンネル3 P+C0H
	E8401AH	R/W	ベース・トランスファ・カウンタ（Word）	
	E8400CH	R/W	メモリ・アドレス・レジスタ（Long Word）	
	E84014H	R/W	デバイス・アドレス・レジスタ（Long Word）	
	E8401CH	R/W	ベース・アドレス・レジスタ（Long Word）	
	；			
	E840FFH	R/W	ゼネラル・コントロール・レジスタ	
エリア・セット	E86001H	W	スーパーバイザ領域設定	
MFP	E88001H	R	GPIP・データ・レジスタ	
	E88003H	W	アクティブ・エッジ・レジスタ	
	E88005H	W	データ・ディレクション・レジスタ	
	E88007H	R/W	割り込みイネーブル・レジスタA	
	E88009H	R/W	割り込みイネーブル・レジスタB	
	E8800BH	R/W	割り込みペンディング・レジスタA	
	E8800DH	R/W	割り込みペンディング・レジスタB	
	E8800FH	R/W	割り込みイン・サービス・レジスタA	

●表２．２　I/Oポート・アドレス一覧（＊は無効）②

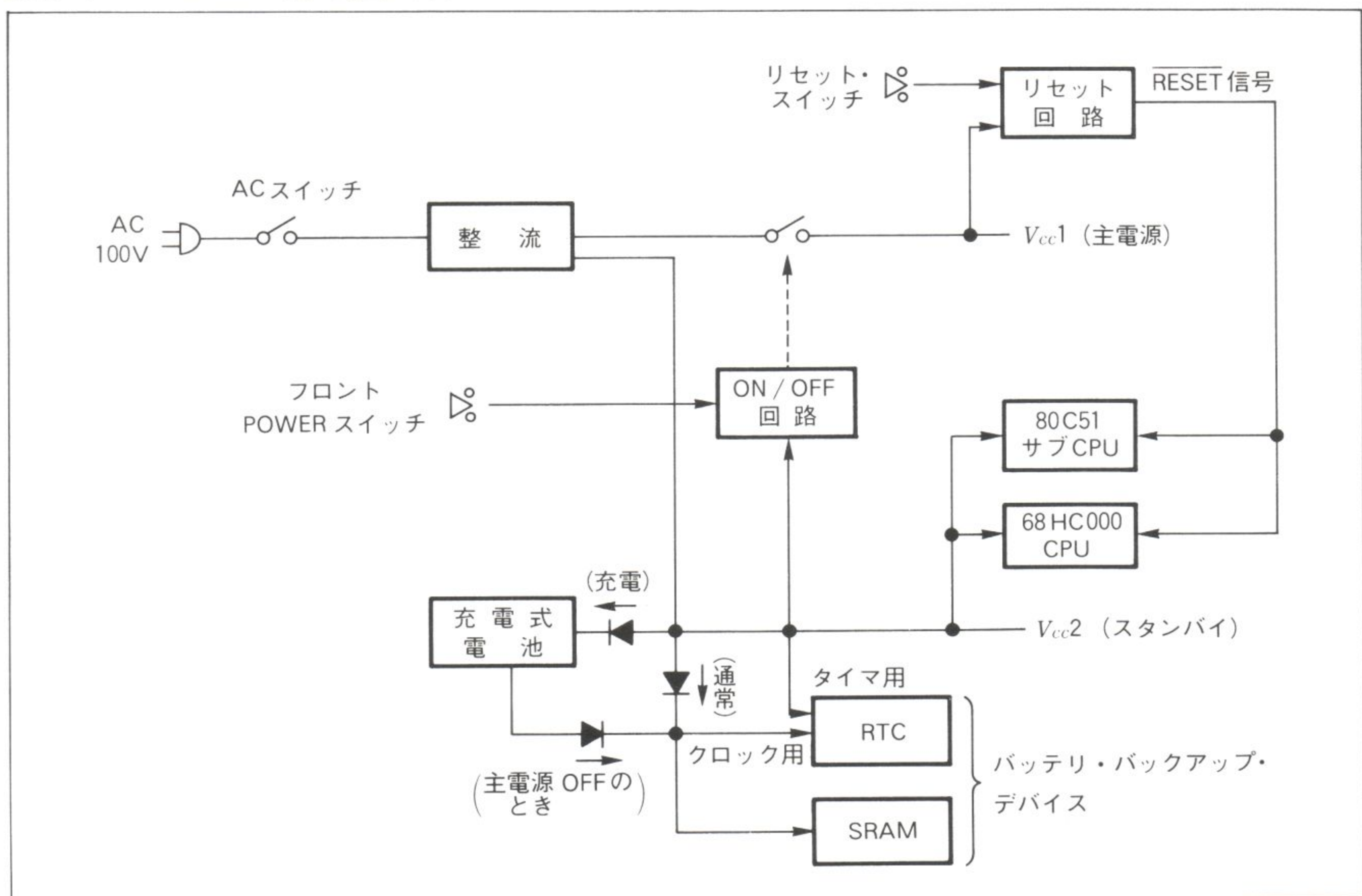
項目	ポート・アドレス	リード/ライト	機能	備考
	E88011H	R/W	割り込みイン・サービス・レジスタＢ	
	E88013H	R/W	割り込みマスク・レジスタＡ	
	E88015H	R/W	割り込みマスク・レジスタＢ	
	E88017H	W	ベクタ・レジスタ	
	E88019H	W	タイマA・コントロール・レジスタ	
	E8801BH	W	タイマB・コントロール・レジスタ	
	E8801DH	W	タイマC/D・コントロール・レジスタ	
	E8801FH	R/W	タイマA・データ・レジスタ	
	E88021H	R/W	タイマB・データ・レジスタ	
	E88023H	R/W	タイマC・データ・レジスタ	
	E88025H	R/W	タイマD・データ・レジスタ	
	E88027H	W	同期キャラクタ・レジスタ	未使用
	E88029H	W	USARTコントロール・レジスタ	
	E8802BH	R/W	受信ステータス・レジスタ	
	E8802DH	R/W	送信ステータス・レジスタ	
	E8802FH	R/W	USART データ・レジスタ	
RTC	E8A001H	R/W	１秒カウンタ/CLKOUTセレクト	
	E8A003H	R/W	10秒カウンタ/Adjust	
	E8A005H	R/W	１分カウンタ/アラーム１分レジスタ	
	E8A007H	R/W	10分カウンタ/アラーム10分レジスタ	
	E8A009H	R/W	１時間カウンタ/アラーム１時間レジスタ	
	E8A00BH	R/W	10時間カウンタ/アラーム10時間レジスタ	
	E8A00DH	R/W	曜日カウンタ/アラーム曜日レジスタ	
	E8A00FH	R/W	１日カウンタ/アラーム１日レジスタ	
	E8A011H	R/W	10日カウンタ/アラーム10日レジスタ	
	E8A013H	R/W	１月カウンタ	
	E8A015H	R/W	10月カウンタ/12・24時セレクタ	
	E8A017H	R/W	１年カウンタ/うるう年カウンタ	
	E8A019H	R/W	10年カウンタ	
	E8A01BH	R/W	モード・レジスタ	
	E8A01DH	W	テスト・レジスタ	
	E8A01FH	W	リセット・コントローラ	
プリンタ	E8C001H	W	プリンタ・データ	
	E8C003H	W	プリンタ・ストローブ	
	E9C001H	R	プリンタ・ビジー	
システムPort	E8E001H	R/W	コントラスト調整（D/A）	
システム・ポート	E8E003H	R/W	TVコントロール	
	E8E005H	W	画像入力コントロール	
	E8E007H	R/W	H/L LED 点灯，NMIリセット，キーコントロール	
	E8E00DH	W	SRAM Write Enable Control	
	E8E00FH	W	POWER OFF Control	
FM音源	E90001H	W	FM音源レジスタ・アドレス・ポート	
	E90003H	R/W	FM音源レジスタ・データ・ポート	
音声合成	E92001H	R/W	ADPCMステータス(IN)/ADPCMコマンド(OUT)	
	E92003H	R/W	ADPCMデータ・レジスタ(IN/OUT)	
	E9A005H	W	ADPCM出力，サンプリング周波数切り換えレジスタ	8255ポートC
フロッピーディスク	E94001H	R	FDCステータス・レジスタ(IN)	
	E94003H	R/W	FDCデータ・レジスタ(IN/OUT)	
	E94005H	R/W	ドライブ・ステータス(IN)/ドライブ・コントロール(OUT)	オプション信号
	E94007H	W	アクセス・ドライブ・セレクト，2HD/2DD・2D切り換え(OUT)	

項目	ポート・アドレス	リード/ライト	機能	備考
ハード・ディスク	E96001H	R/W	HDデータ(IN/OUT)	
	E96003H	R/W	ステータス(IN)/セレクト・リセット(OUT)	
	E96005H	W	コントローラ・ボード・リセット(OUT)	
	E96007H	W	セレクト・セット(OUT)	
S C C	E98001H	R/W	SCCコマンド・ポートB	
	E98003H	R/W	SCCデータ・ポートB	
	E98005H	R/W	SCCコマンド・ポートA	
	E98007H	R/W	SCCデータ・ポートA	
ジョイスティック	E9A001H	R	ジョイスティック　0	8255ポートA
	E9A003H	R	ジョイスティック　1	8255ポートB
8255	E9A007H	W	8255 コントロール・ワード・レジスタ	
FD, PR, HD	E9C001H	R/W	FDC, FDD, HD, プリンタ 割り込みステータス (IN) FDC, FDD, HD, プリンタ 割り込みマスク(OUT)	
FD, PR	E9C003H	W	FDC, FDD, HD, プリンタ 割り込みベクタ	

2-2 電源系統の構成

X68000の電源系統の概念図を図2.3に示します．

内部の電源線は3系統に分けられ，主電源(**Vcc1**)のほかにスタンバイ用の**Vcc2**，**バッテリ・バックアップ**用の充電式電池が装備されています．

●図2．3　X68000の電源系統概念図

Vcc2は，背面のACスイッチが入っている間供給され，68HC000CPUなどに接続されているため，主電源がONでなくてもTVコントロールが行なえます．またこの電源は充電式電池に接続され，充電にも利用されます．

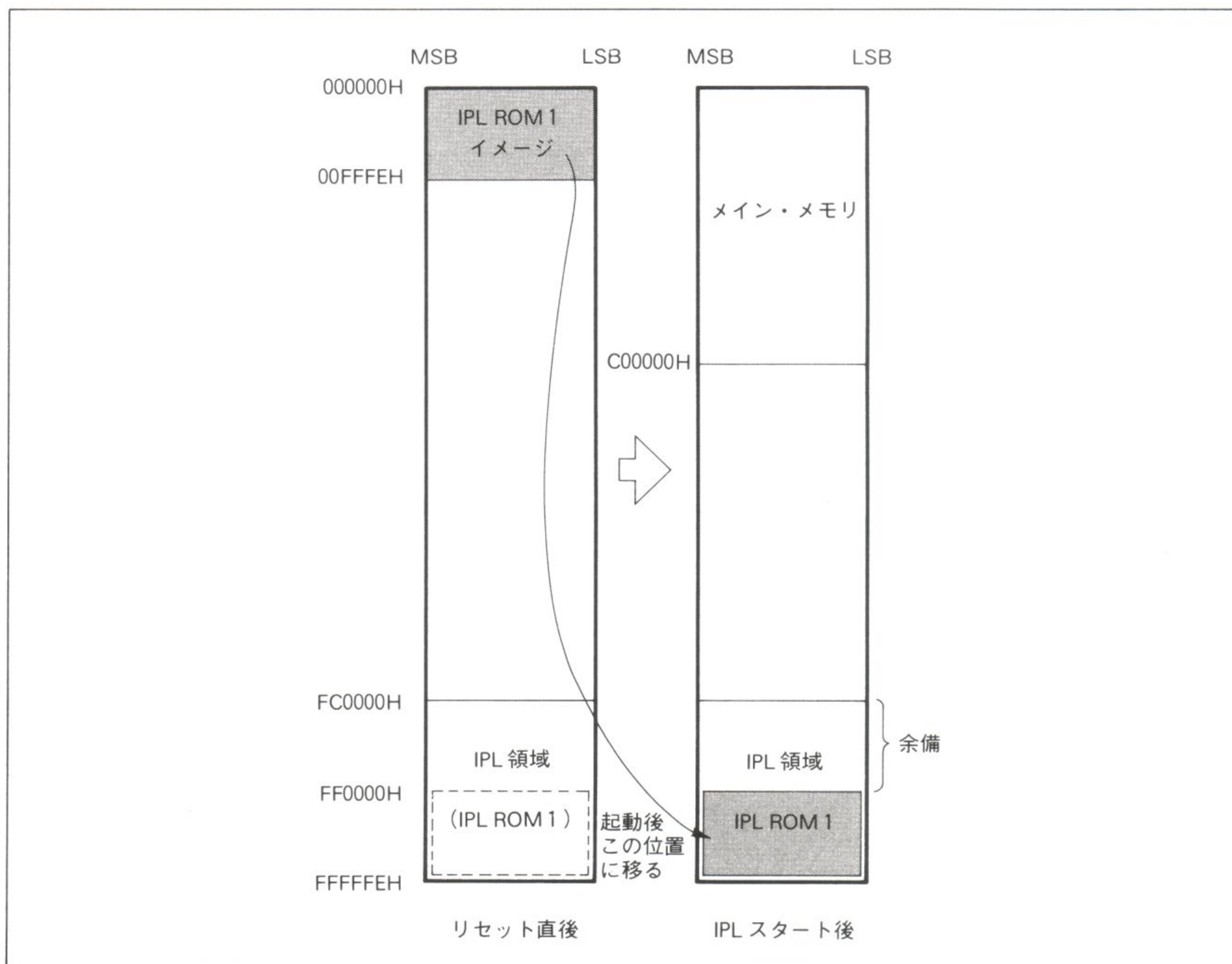
AC電源を切ったときでも，時刻やデータの保持は止めるわけにはいきません．このため，RTC（リアル・タイム・クロック）やメモリ・スイッチ用のSRAM（スタティックRAMに使われるバッテリ）は，AC電源が供給されないときは放電して，これらのデバイスのバックアップを行ないます．このときは低電圧駆動となり，消費電力が大幅に少なくなるので，かなり長期間通電しなくても大丈夫です．

リセット回路は主電源と連動しており，POWER ON により自動的にリセット信号($\overline{\text{RESET}}$)を発生して，68000を起動状態にさせます．リセットは，本体上部の**リセット・スイッチ**によっても発生させることができます．

主電源をON／OFFするスイッチは本体に内蔵されていて，フロントの**POWERスイッチ**がONの場合のみ連動します．OFFの場合は，POWERスイッチがOFFであることだけでなく，アドレスE8E00F（POWER OFFコントロール・ポート）に00H，0FH，0FHの手順で書き込みを行なわないとOFFにできないようになっています．これはディスクが動作中に誤ってOFFにすることを防止するための仕組ですが，ACスイッチが切られると強制OFFとなって，安全が保たれません．したがって，ACスイッチを切る際は，主電源のOFFを確認した上で行なうよう注意する必要があります．

2-3 ROMによるシステムの起動

電源がON，またはリセット・スイッチが押されたとき，68000CPUにはリセット信号が供給され，CPU

●図2．4　リセット時のメモリ編成の変化

の立ち上げ動作が開始されます．

リセット時にはシステム・ベクトル領域のうち**SSP初期値**，**PC初期値**が最初に読み取られるので，最小限これだけの値を与えてやらないと起動することができません．

また，PC初期値によって実行開始されるプログラムもさしあたりどうしても必要なので，1つのROMの中にこれらの内容を焼き付けてあります．ただし，システム・ベクトル値は一度読み込まれると，次のリセット時まで再度読み込まれることはないので，プログラムの実行が開始されると，0番地付近の内容は撤退するようになっています．この様子は，図2.4に示すとおりで，撤退後の0番地からの内容は，メイン・メモリ(RAM)に置き換わります．

ROMによって最初に実行されるプログラムは，**IPL**(イニシャル・プログラム・ローダ)と呼ばれ，メモリ・スイッチを参照して所定のディスク・ドライブからモニタ・プログラムをロードし，完了すると，モニタ・プログラムに処理を引き継ぐようになっています．

X68000では，アドレスFF0000～FFFFFFにわたって，IPL-ROMが配置され，その下のFC0000～FEFFFFが，IPL-ROMの増設エリアとして空けられています．

2-4 システム・ポート

システム・ポートは，基本的には周辺装置に属さない，システムの基本的な諸動作を制御するポートで，たとえばPOWER OFFとか，SRAMの書き込み禁止および解除といった動作を行ないます．表2.3はシステム・ポートのアドレス・マップを示したもので，この中には周辺装置との関連が強いので，一般のI/Oポートに分類したほうがよさそうなもの(たとえばコントラストなど)もありますが，コンソール・デバイスという広義の意味で，メーカーではシステム・ポートに含めたものかも知れません．

個々のポートの働きについては，備考またはポートの ビット構成(表2.4)に説明してあります．

●表2．3　システム・ポート・レジスタ・アドレス・マップ

No	レジスタ・アドレス	R/W	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備考
1	E8E001H	R/W					コントラスト調整				0H(暗)～FH(明)の16階調
2	E8E003H	R/W					TVコントロール	FIELD (READ) のみ	3D L	3D R	
3	E8E005H	W				画像入力コントロール					ディジタイズテロッパ（オプション）用
4	E8E007H	R/W					キーコントロール	NMIリセット	HRL		
5	E8E00DH	W	SRAM Write Enable Control								31Hを書くと書き込み許可
6	E8E00FH	W					POWER OFF Control				

●表2．4　システム・ポート・ビットの内訳

E8E003H
〔READ/WRITE〕

Bit	WRITE	READ	
D_0	3D　R	3D　R	立体視装置（オプション）0：シャッターOPEN 1：シャッターCLOSE
D_1	3D　L	3D　L	
D_2		FIELD(液晶)	0：1番，1：2番
D_3	TVリモコン信号	TV ON/OFFステータス	

E8E007H
〔READ/WRITE〕

Bit	WRITE	READ	備　　考
D_1	HRL	HRLステータス	予約ずみ
D_2	NMIリセット	――――	NMIルーチンで1を書いてからRTI命令を実行する
D_3	キーレディ（データ送信許可時1）	キージャックステータス（ジャック挿入時1）	

2-5 ハードウェア割り込み

68000の一般のハードウェア割り込みには，優先度が与えられており，1～7にレベル付けされた割り込み入力線で受信されるようになっています．表2.5は，各レベルと周辺デバイスなどとの対応を一覧にしたもので，NMIスイッチ(本体上部のINTERRUPT)のものが最高レベル，フロッピー，プリンタが最低レベルとなっています．

レベル7は最高レベルであるとともに，マスクできない割り込みで，いわば緊急避難用です．たとえばプログラムのアボートに失敗したときなどは，何度再トライしてもプログラム終了まで止まらない*ものです．このようなときは，リセットしてやり直すまでもなく，NMIスイッチを押せば強制的に止めることができます．

レベル6のMFP(マルチ・ファンクション・ペリフェラル)にはたくさんの周辺デバイスが接続されていて，MFPにより表2.6の優先度に従った割り込みの制御が行なわれます．すなわち，MFPは優先度の高いものから順に，68000に対し割り込みの中継をします．

●表2．5　68000の割り込み

レベル	割り当て	要　　因
高 7	NMI	外部NMIswによる割り込み　（オートベクトル割り込み）
6	MFP	各種タイマ，KEYデータ受信，H-SYNC，V-DISPなどによる割り込み　（ベクトル割り込み）
5	SCC	RS-232C，マウスデータ受信による割り込み　（ベクトル割り込み）
4	―	拡張I/Oスロット
3	DMAC	転送終了などによる割り込み　（ベクトル割り込み）
2	―	拡張I/Oスロット
低 1	フロッピー，プリンタ	FDC，FDD，ハードディスク，プリンタBUSYなどによる割り込み（ただし，FDC>FDD>HD>プリンタの順で優先順位が構成されている）　（ベクトル割り込み）

＊ この現象は，後のバージョンで解決される可能性もある．

NMI スイッチによる割り込み以外の割り込みは，オートベクトルではないので，表2.7のように前もってベクトルを設定しておかなければなりません．このうち MFP については上位 4 ビットのみの設定で，後は MFP 内部の優先順位が下位 4 ビットとして使われます．

●表2．6　MFP（マルチ・ファンクション・ペリフェラル）の割り込み，読み出しポート

優先順位	チャンネル	要因	一般名称
高 15	1111	CRTC からの H-SYNC 信号	General Purpose Interrupt 7(17)
14	1110	CRTC からの IRQ 信号（任意の H ラスターに指定可）	General Purpose Interrupt 6(16)
13	1101	CRTC からの V-DISP 信号	Timer A
12	1100	KEY データの受信割り込み	Receiver Buffer Full
11	1011	KEY データの受信エラー	Receive Error
10	1010	KEY データの送信割り込み	Transmit Buffer Empty
9	1001	KEY データの送信エラー	Transmit Error
8	1000	USART（キーボード）シリアル・クロック	Timer B
7	0111	未使用	
6	0110	CRTC の V-DISP 信号の状態検出	General Purpose Interrupt 4(14)
5	0101	8 ビット汎用タイマ（入力クロック 4 MHz）	Timer C
4	0100	8 ビット汎用タイマ（入力クロック 4 MHz）	Timer D
3	0011	FM 音源による割り込みの検出	General Purpose Interrupt 3(13)
2	0010	POWER SW による ON/OFF の検出	General Purpose Interrupt 2(12)
1	0001	拡張用10スロットからの EXPWON 信号による ON/OFF の検出	General Purpose Interrupt 1(11)
低 0	0000	RTC の ALARM 信号による ON/OFF の検出	General Purpose Interrupt 0(10)

●表2．7　割り込みベクトルの設定

割り当て	レジスタ・アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
MFP	E88017H	設定（D_7～D_4）				0—自動割り込み終了モード 1—ソフトウェア割り込み終了モード			
SCC (チャンネルA, B共用)	書き込みレジスタ2に設定	設定（D_7～D_0）							
DMAC (チャンネル0) (内蔵2HD)	E84025H (ノーマル・インタラプト・ベクタ)	設定（D_7～D_0）							
	E84027H (エラー・インタラプト・ベクタ)	設定（D_7～D_0）							
DMAC (チャンネル1) (ハードディスク)	E84065H (ノーマル・インタラプト・ベクタ)	設定（D_7～D_0）							
	E84067H (エラー・インタラプト・ベクタ)	設定（D_7～D_0）							
DMAC (チャンネル2) (メモリ-メモリ)	E840A5H (ノーマル・インタラプト・ベクタ)	設定（D_7～D_0）							
	E840A7H (エラー・インタラプト・ベクタ)	設定（D_7～D_0）							
DMAC (チャンネル3) (音声合成)	E840E5HE (ノーマル・インタラプト・ベクタ)	設定（D_7～D_0）							
	E840E7H (エラー・インタラプト・ベクタ)	設定（D_7～D_0）							
フロッピー, プリンタ	E9C003H	設定（D_7～D_2）						FDC割り込み—0 FDD割り込み—0 ハードディスク割り込み—1 プリンタ割り込み—1	0 1 0 1

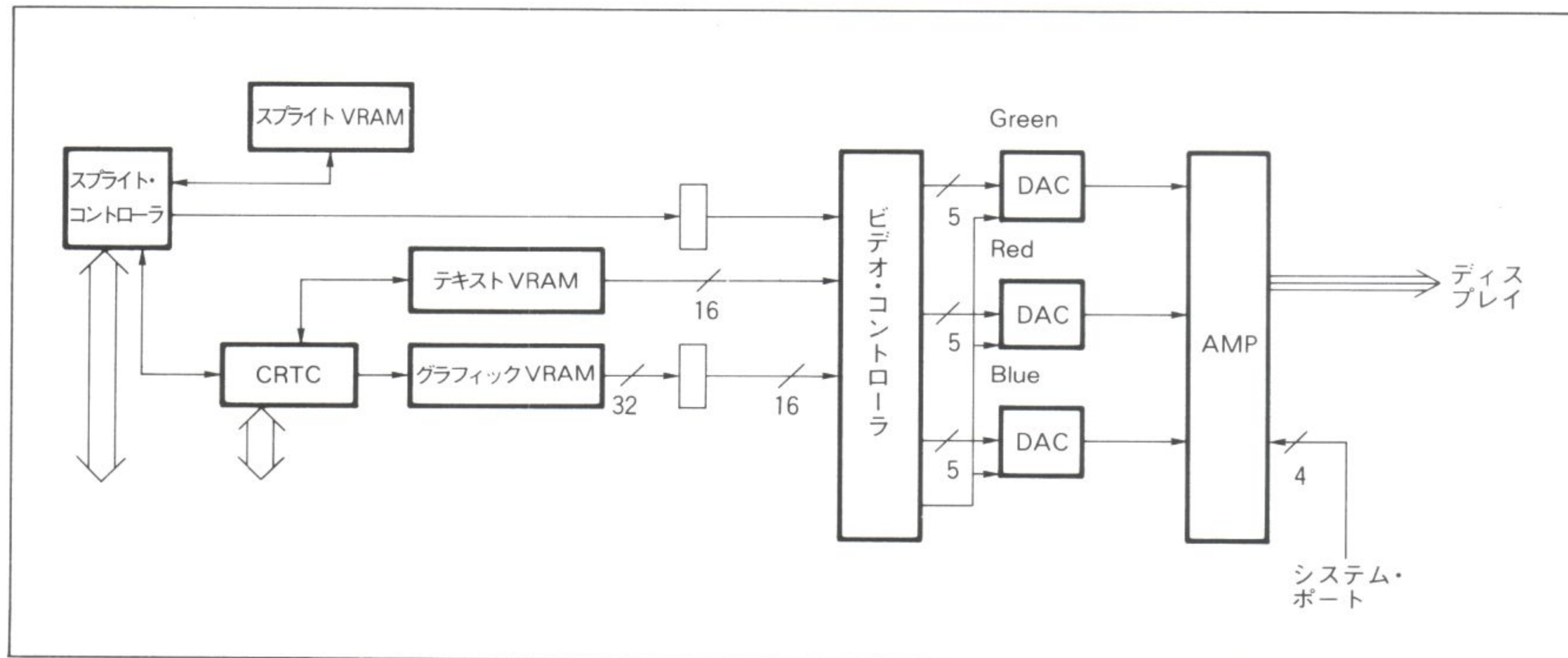
第3章

画面制御のハードウェア

3-1 画面制御系の概要

X68000では，68000本来の高速描画(演算)能力に加えて，強力な**スプライト**のサポートにより，高度なゲーム・プログラムのような動きの激しい画面に対応できるようになっています.

一般に画面制御系は，中心となる**CRTC**(CRT コントローラ)，画像データをもつ**リフレッシュ RAM**(VRAM)，ドット単位に転送するシフト・レジスタなどを含んだ**ビデオ・コントロール回路**の3ブロックから構成されます．X68000の場合，スプライト RAM をもっているため，スプライト・コントローラが介在して，図3.1のようなブロック構成となっています.

●図3．1　画面制御系ブロック図

ほとんどのパソコンにとって，画面の制御は大きなウェイトを占めています．そしてこの部分は，機種ごとにユニークにならざるを得ない部分であることも否定できません．

言い換えれば，高速グラフィックスを扱うプログラムは，どうしても機種に依存したものになってしまうのが実態で，それほどハードウェアの構造を意識してかからないと，満足できるものが作れません．

そのような観点から本章では，周辺装置のうちでも画面制御だけをとくに独立させて扱うことにしました．X68000の場合，スプライトなどの画像処理がいわば「売り物」であるだけに，なおさら詳しい情報が必要であると考えられたからです．

したがってリフレッシュ RAM は**テキスト VRAM**，**グラフィック VRAM** のほかに**スプライト VRAM** の3種類に分かれ，これらの読み出し結果がビデオ・コントローラで合成され，ドット単位に出力されます．らにドット情報は DAC(ディジタル──→アナログのコンバータ)でアナログ化され，AMP を経てディスプレイに送られます．

ディスプレイに入る信号は，アナログであるため，可能性としては無限の階調が選べます．しかしディジタル系の精度が65,536色なので，これがシステムの限界となります．

画面の分解能は，高解像度がラスター数512本，低解像度が同256本またはインタレース*方式で512本となっています．

68000では1画素がメモリ1ワードに対応するシンプルな設計となっており，ドットに分解するシフト・レジスタを必要としないことが大きな特徴です．

---

*偶数，奇数番目のラスターを画面ごとに交互に表示し見かけ上の本数を増やす方法．

3-2 テキスト画面のハードウェア構成

コマンドのエコーバックなど，文字列を表示するテキスト画面用のハードウェアは，表3.1の仕様で設計されています．ここで，高解像度モードで256ラインの表示を行なわせる際は，実際は512ラインのままで，2回同一行を読んで転送することにより結果的に256ラインに見せる方法をとっています．

テキスト画面のリフレッシユ・メモリ(VRAM)は，図3.2のように4画面分用意されており，スクリーン(仮想画面)との対応関係を考慮して描き直すと図3.3のように表わすことができます．

表示色は16色ですが，**パレット**を使って変換すれば，65,536色に対応できます．この場合，スクリーンに同時に表示できるのは，あくまで65,536色中の任意の16色となることは言うまでもありません．パレット・レジスタのアドレスは，テキスト画面番号に対応するビットを4個組み合わせたものに対して，表3.2のように割り付けられています．

テキスト画面のスクロールは**円筒スクロール**で，左右方向の端は切れていますが，上下方向の端はつながっていて，あたかも連続しているかのように動作します．

●表3．1　表示画面構成

仮想画面(HxV)	同時表示色(パレット色)	同時表示面数(重ね合わせ面数)	表示画面(HxV)	備考
1,024×1,024	16(65,536)	1	高解像度モード 768×512 512×512 512×256 256×256	 2度読み 2度読み
			低解像度モード(オーバースキャン) 512×512 512×256 256×256	•オーバースキャンのみスーパーインポーズ •オーバースキャンのため実際の表示画面サイズは小さくなる インターレース

●図3．2　テキストVRAMのメモリ・マップ

●図３．３　テキスト仮想画面のアドレス配置

1,024ドット
ワード
0 〜 63

T3プレーン
E60000　E6007E
E7FFFE

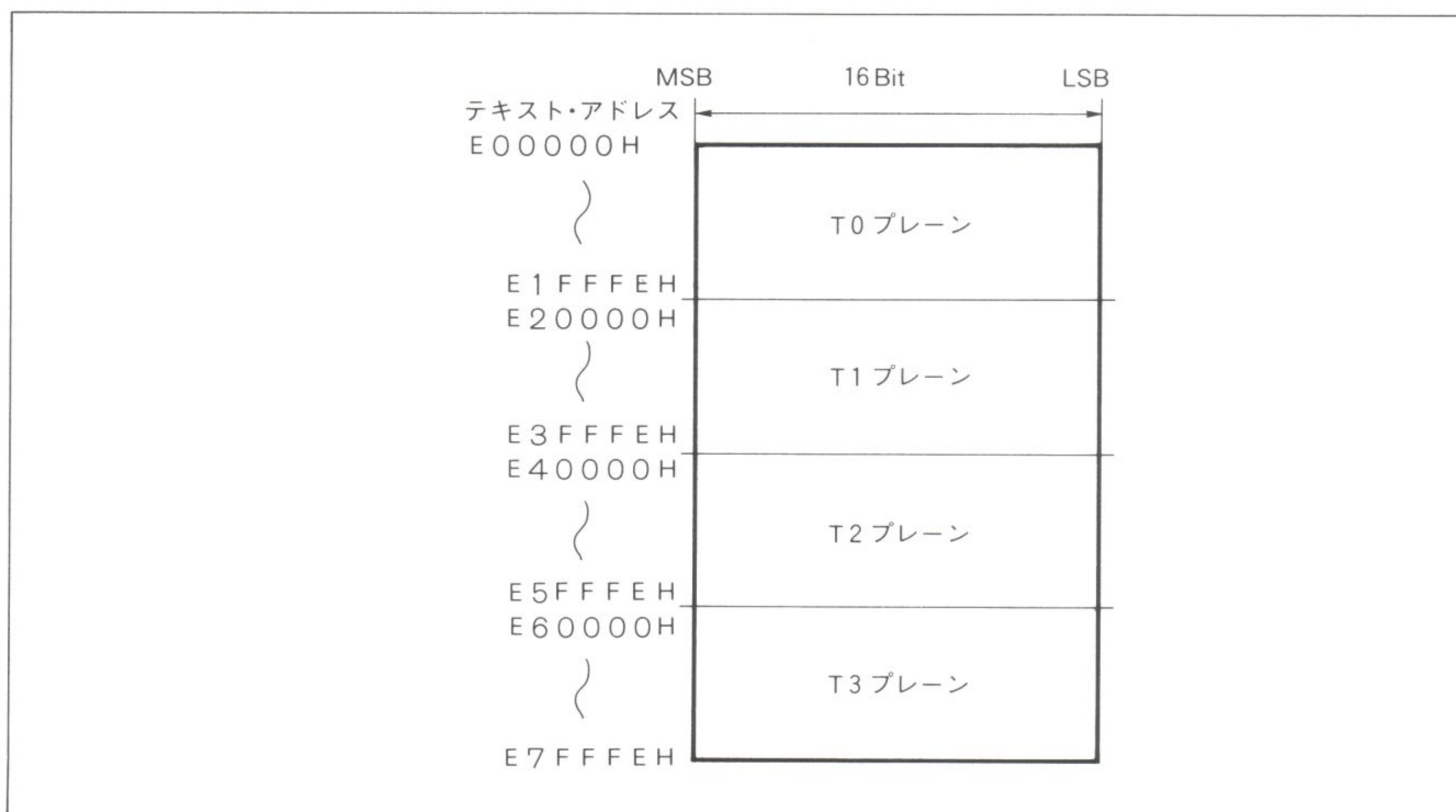
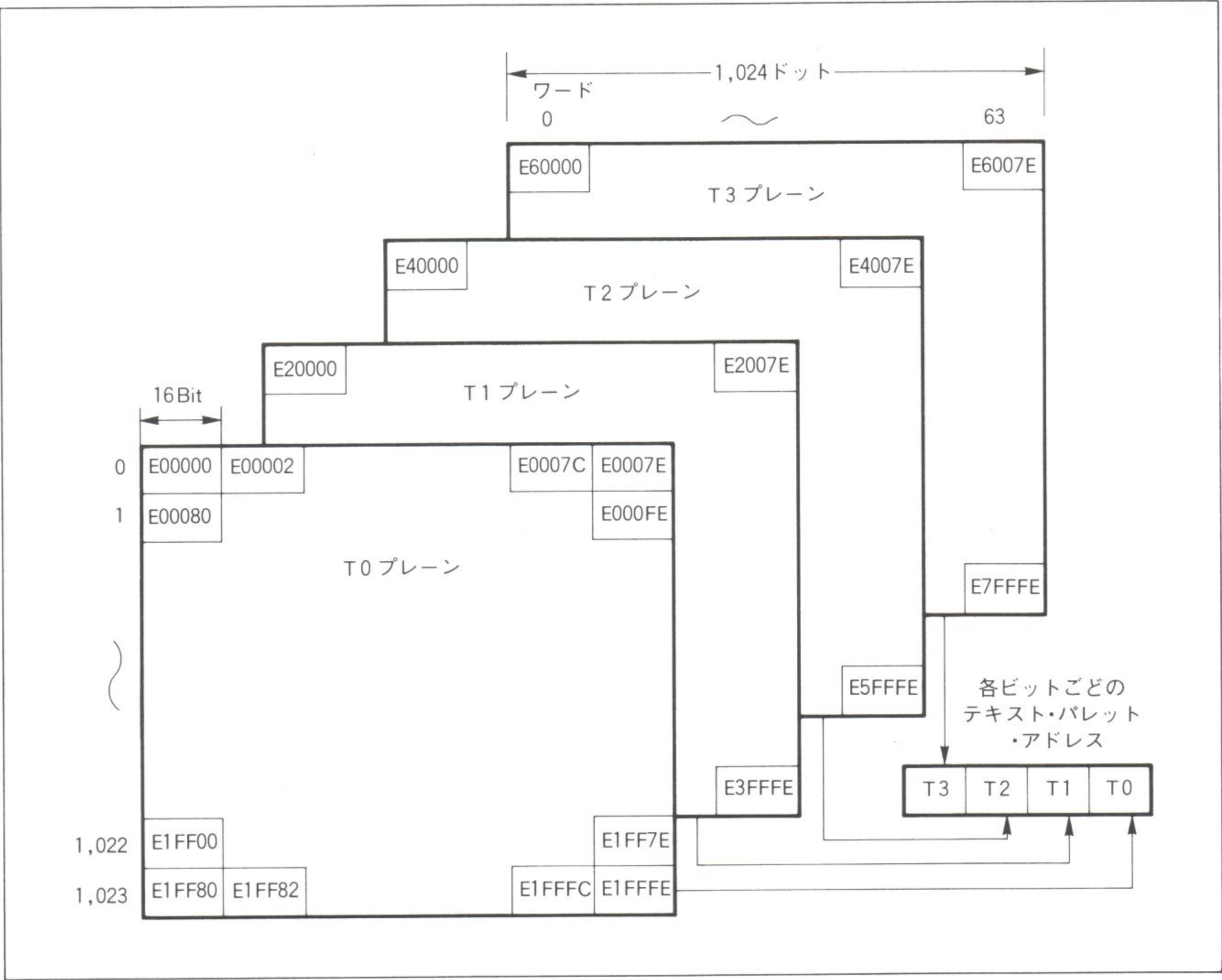
T2プレーン
E40000　E4007E
E5FFFE

T1プレーン
E20000　E2007E
E3FFFE

16Bit

T0プレーン
0　E00000　E00002　E0007C　E0007E
1　E00080　E000FE
1,022　E1FF00　E1FF7E
1,023　E1FF80　E1FF82　E1FFFC　E1FFFE

各ビットごとの
テキスト・パレット
・アドレス

T3　T2　T1　T0

●表３．２　テキスト・パレット・アドレス［READ/WRITE 可］

パレット・アドレス	レジスタ・アドレス	D_{15}—D_{11}	D_{10}—D_{6}	D_{5}—D_{1}	D_{0}	備　　考
		Green	Red	Blue	I	
00H	E82200H	16bitデータ				ただし，このパレット・アドレスは，スプライト・カラー０のパレットと共通に使用するものとする．
01H	E82202H					
02H	E82204H					
03H	E82206H					
∫	∫					
0CH	E82218H					
0DH	E8221AH					
0EH	E8221CH					
0FH	E8221EH					

3-3 グラフィック画面のハードウェア構成

グラフィック画面では，VRAMの編成の仕方により4通りの表示形態を使用することができます(表3.3)．これは，メモリを，1,024×1,024ドット×16(2^4)色(①)で使用するのと，512×512ドット×16(2^4)色×4画面(②)の構成にするのでは容量が同じで，後者の場合256(2^8)色×2画面(③)，65,536(2^{16})色×1画面(④)でもメモリ・サイズは変わらないためです．

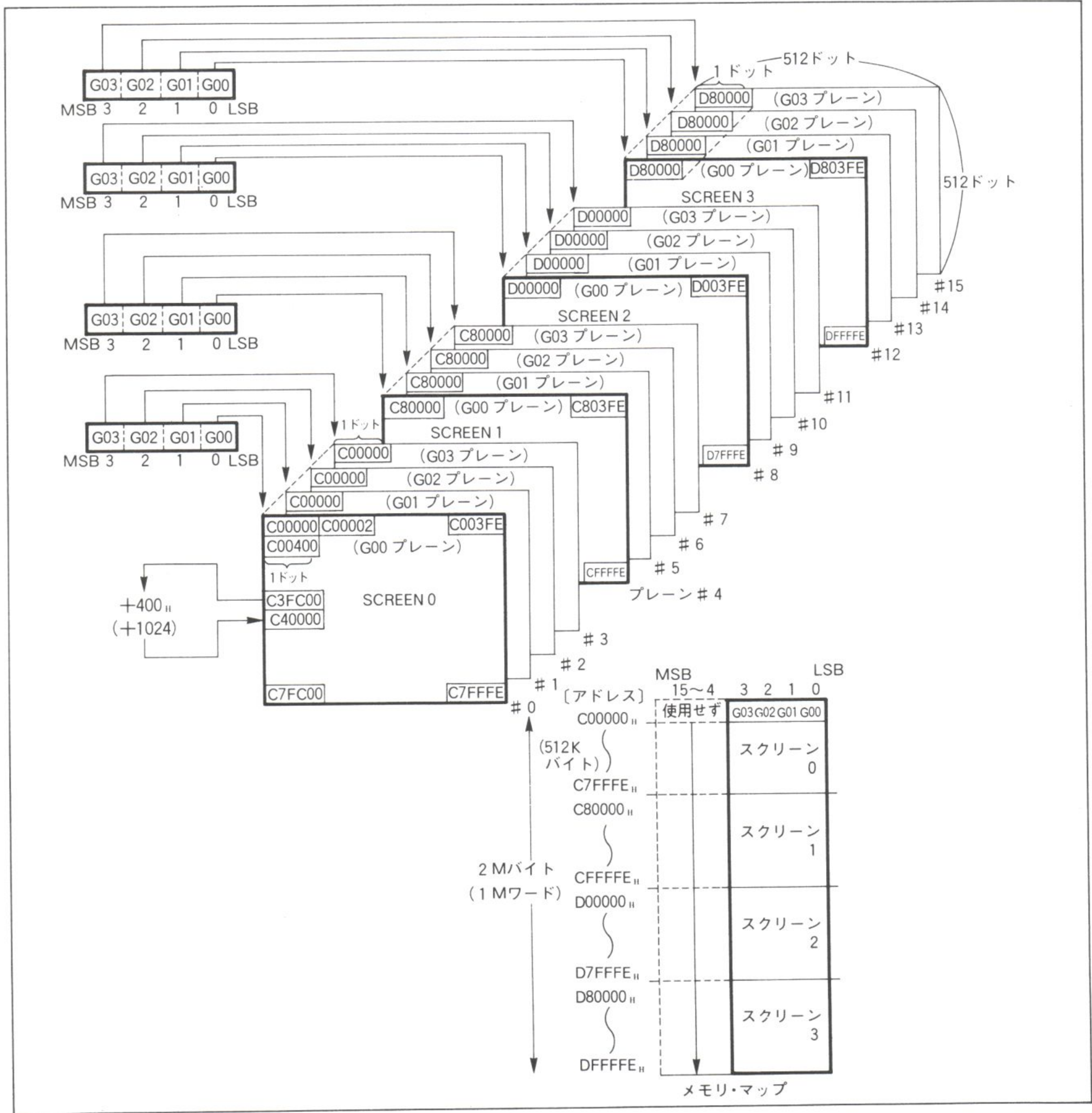
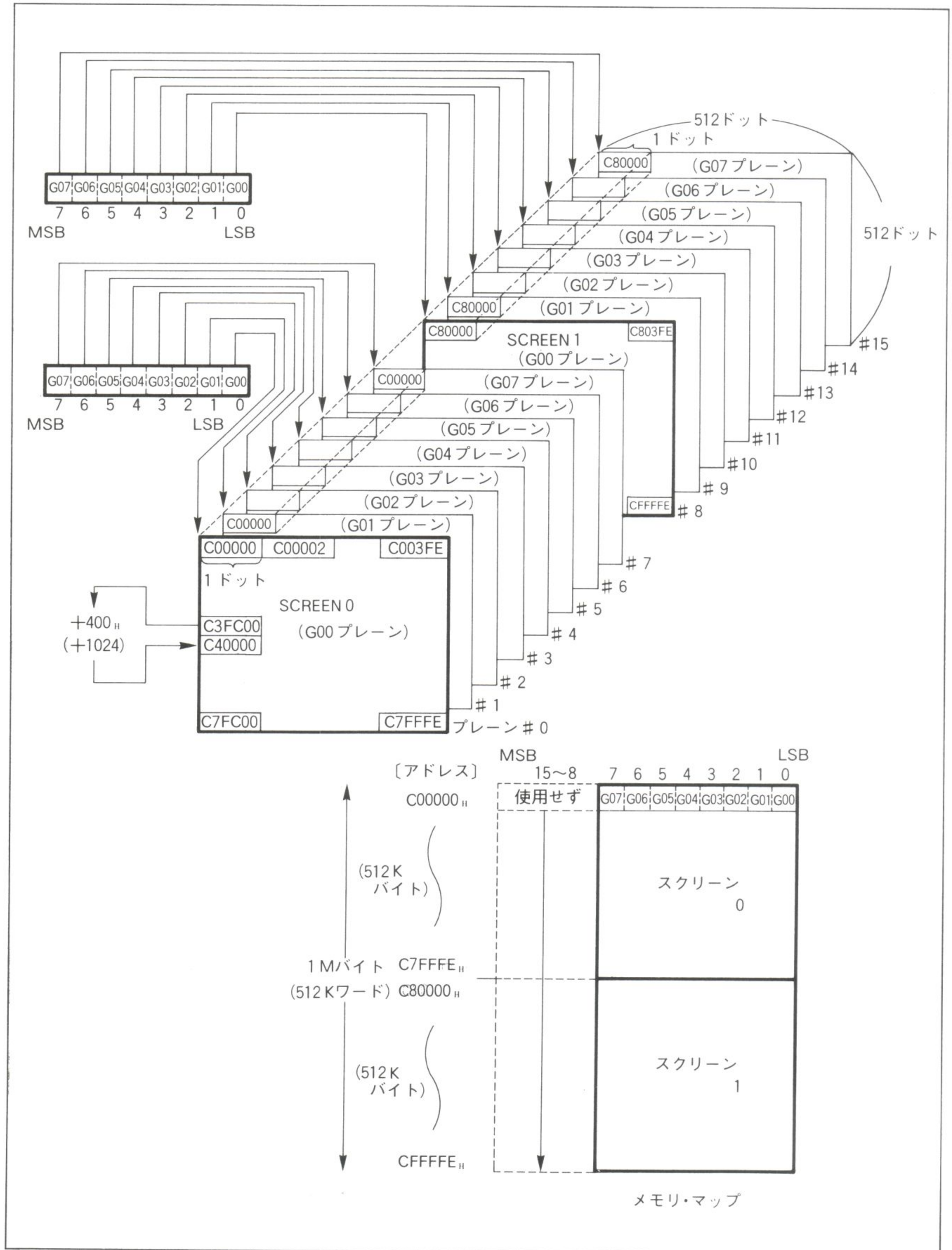
実際のメモリの構成と仮想画面との対応は，①が図3.4，②が図3.5，③が図3.6，④が図3.7にそれぞれ対応します，いずれも，メモリのデータが色情報であり，各ビットが各画面のカラーコードを直接表わしますが，パレットの使用により65,536色中任意の色(色数の限度はそれぞれの表示色数で決まる)に変換できます．

こういったメモリ構成では，同一ワードに複数のドット情報をもたないため，演算による描画の際にプ

●図3．4　グラフィック仮想画面のアドレス配置：1,024×1,024ドット（16色モード）

●表3．3　表示画面構成

仮想画面 (HxV)	同時表示色 (パレット色)	同時表示面数 (重ね合わせ面数)	表示画面 (HxV)	備　考
1,024 ×1,024	16 (65,536)	1	高解像度モード 768×512 512×512 512×256 256×256	 ２度読み ２度読み
			低解像度モード (オーバースキャン) 512×512 512×256 256×256	•オーバースキャンのみスーパーインポーズ •オーバースキャンのため実際の表示画面サイズは小さくなる インターレース
512×512	16 (65,536)	4	高解像度モード 512×512 512×256 256×256	 ２度読み ２度読み
	256 (65,536)	2		
	65,536 (65,536)	1	低解像度モード (オーバースキャン) 512×512 512×256 256×256	•オーバースキャンのみスーパーインポーズ •オーバースキャンのため実際の表示画面サイズは小さくなる インターレース

●図3．5　グラフィック仮想画面のアドレス配置：512×512ドット（16色，４面モード）

●図3．6　グラフィック仮想画面のアドレス配置：512×512ドット（256色，2面モード）

●図3．7　グラフィック仮想画面のアドレス配置：512×512ドット（65,536色，1面モード）

G15 G14 G13 G12 G11 G10 G09 G08 G07 G06 G05 G04 G03 G02 G01 G00
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
MSB
〔データ構成〕

512ドット
C00000　(G15プレーン)
(G14プレーン)
(G13プレーン)
(G12プレーン)
(G11プレーン)
(G10プレーン)
(G09プレーン)
(G08プレーン)
(G07プレーン)
(G06プレーン)
(G05プレーン)
(G04プレーン)
(G03プレーン)
(G02プレーン)
(G01プレーン)
512
ドット
#15 #14 #13 #12 #11 #10 #9 #8 #7 #6 #5 #4 #3 #2 #1

C00000　C00002　C003FE
1ドット　(G00プレーン)
+400 H
(+1024)
C3FC00
C40000
C7FC00　C7FFFE　プレーン#0

〔アドレス〕
MSB　LSB
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
C00000 H　G15 G14 G13 G12 G11 G10 G09 G08 G07 G06 G05 G04 G03 G02 G01 G00
512Kバイト
(256Kワード)
C7FFFE H
メモリ・マップ

ログラムを簡略化，高速化できます．すなわち，複数ドットを同居させる方法では，ハードウェアは簡単になるのですが，転送時に関係のないドット・データをマスクするなど余分な処理が必要なため，どうしてもプログラムが複雑化してしまうのです．

パレットは，各表示形態とも同じレジスタを使います．したがって16色，256色ともに表3.4のような構成になりますが，65,536色では少し使い方が異なり，表3.5の割り付けが適用されます．この表はちょっと変わっていて，パレット・アドレス上位，下位バイトの値により参照するビット範囲が異なります．すなわち，0302H ならばレジスタ・アドレス E82006H の下位 8 ビットが，パレット値上位 8 ビットとなり，E82004H の上位 8 ビットがパレット値下位 8 ビットとして出力されます．いわば表の上半分がパレット値下位 8 ビットであり，下半分が同上位 8 ビットを表わします．また，左半分がパレット・アドレス偶数時，右半分がパレット・アドレス奇数時に対応します．

グラフィック画面のスクロールは，**球面スクロール**で，上下，左右ともに端が連続しているように動作します．

●表3．4　グラフィック・パレット・アドレス
グラフィック16色，256色モード［READ/WRITE 可］

パレット・アドレス VRAMデータ	レジスタ・アドレス	D_{15}—D_{11}	D_{10}—D_6	D_5—D_1	D_0	備考
		Green	Red	Blue	I	
00H 01H 02H 03H ↓ 0CH 0DH 0EH 0FH	E82000H E82002H E82004H E82006H ↓ E82018H E8201AH E8201CH E8201EH	←16 bit データ→				16色モード・パレット 256色モード・パレット
10H 11H 12H ↓ FDH FEH FFH	E82020H E82022H E82024H ↓ E821FAH E821FCH E821FEH	↓				

●表3．5　グラフィック・パレット・アドレス
グラフィック65,536色モード［READ/WRITE 可］

パレット・アドレス グラフィックVRAM メモリ・データ 下位バイト	レジスタ・アドレス	D_{15} D_{14}	D_{13}—D_9	D_8	パレット・アドレス グラフィックVRAM メモリ・データ 下位バイト	D_7 D_6	D_5—D_1	D_0	備考
		Red 下位	Blue	I		Red 下位	Blue	I	
00H 02H 04H ∫ FCH FEH	E82000H E82004H E82008H ∫ E821F8H E821FCH	←8 bit→			01H 03H 05H ∫ FDH FFH	←8 bit→			下位バイトの値により65,536色モード・パレット，D_{15}—D_8の 8 bitか，D_7—D_0の 8 bitをセレクトし，下位 8 bitとして出力する

パレット・アドレス グラフィックVRAM メモリ・データ 上位バイト	レジスタ・アドレス	D_{15}—D_{11}	D_{10}—D_8	パレット・アドレス グラフィックVRAM メモリ・データ 上位バイト	D_7—D_3	D_2—D_0	備考
		Green	Red 上位		Green	Red 上位	
00H 02H 04H ∫ FCH FEH	E82002H E82006H E8200AH ∫ E821FAH E821FEH	←8 bit→		01H 03H 05H ∫ FDH FFH	←8 bit→		上位バイトの値により65,536色モード・パレット，D_{15}—D_8の 8 bitか，D_7—D_0の 8 bitをセレクトし，上位 8 bitとして出力する

＊グラフィック画面同士の半透明処理を行なうときは，偶数・奇数パレットの内容を同じにする．

3-4 CRTCのレジスタとその設定の仕方

X68000では，専用CRTCを使い，各種のモード，表示形態に対応しています．表3.6は主として同期信号を中心に示したCRTCの仕様で，解像度に対応した時間値が規定されています．

内部のレジスタは表3.7の構成になっており，このうち$R_{0\sim8}$は表3.8のように設定します．R_9以下のレジスタの説明と設定方法は，図3.8のとおりです．

これらのレジスタのうち，**動作設定ポート**は**ラスター・コピー**，グラフィックVRAM**高速クリア**，**画像入力**に使われ，書き込みはいわばコマンド発信をすることになります．また，読み出しは，該当機能が実行中であるかどうかのステータスを参照することを意味し，実行中ならば1が得られます．

設定内容が複雑なレジスタはR21で，高速クリア時に下位4ビットの値の意味が表示形態ごとに異なります．また，それに伴って高速クリアされる範囲も微妙に違ってくるので，注意が必要です．

レジスタの設定順序は，表示密度の高いものから低いものに変更するときと，逆の場合とでは，内部のカウント・アップ動作の都合から異なったシーケンスが規定されています．すなわち，高──→低の場合はR20──→R1……R7──→R0，低──→高の場合はR0──→R1……R7──→R20順です．どちらが高いレベルにあるかは，R20に設定する下位ビットの値で決まります．

●表3．6　CRTC仕様

表示モード		高解像度	低解像度
走査方式		ノンインターレース	ノンインターレース,インターレース
同期周波数	水平(KHz) 垂直(Hz)	31.5 55.46	15.98 61.46
データ表示期間（1）	水平(μsec) 垂直(msec)	22.09 16.25	52.69 15.019
同期期間（2）	水平(μsec) 垂直(msec)	31.75 18.03	62.58 16.270
同期パルス幅（3）	水平(μsec) 垂直(msec)	3.45 0.191	3.30 0.187
バックポーチ（4）	水平(μsec) 垂直(msec)	4.14 1.111	4.94 0.876
フロントポーチ（5）	水平(μsec) 垂直(msec)	2.07 0.476	1.65 0.187

・映像信号　アナログ0.7Vp-p（75　終端）正極性
・同期信号　TTLレベル負極性

●表3．7　CRTC レジスタ・アドレス・マップ

レジスタNo.	レジスタ・アドレス	リード/ライト	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_{9}	D_{8}	D_{7}	D_{6}	D_{5}	D_{4}	D_{3}	D_{2}	D_{1}	D_{0}	備考
R00	E80000H	W									水平トータル−1								水平同期信号部 8ドット単位 (ただしR00のLSBは無効)
R01	E80002H	W									水平同期終了位置−1								
R02	E80004H	W									水平表示開始位置−1								
R03	E80006H	W									水平表示終了位置−1								
R04	E80008H	W							垂直トータル−1										垂直同期信号部 ラスター単位
R05	E8000AH	W							垂直同期終了位置−1										
R06	E8000CH	W							垂直表示開始位置−1										
R07	E8000EH	W							垂直表示終了位置−1										
R08	E80010H	W									外部同期水平アジャスト								水平位置微調整
R09	E80012H	W							ラスター割り込み位置										ラスター・アドレス
R10	E80014H	W							X方向スクロール										テキスト・スクロール・レジスタ
R11	E80016H	W							Y方向スクロール										
R12	E80018H	W							スクリーン0　X										グラフィック・スクロール・レジスタ
R13	E8001AH	W							スクリーン0　Y										
R14	E8001CH	W								スクリーン1　X									
R15	E8001EH	W								スクリーン1　Y									
R16	E80020H	W								スクリーン2　X									
R17	E80022H	W								スクリーン2　Y									
R18	E80024H	W								スクリーン3　X									
R19	E80026H	W								スクリーン3　Y									
R20	E80028H	R	0	0	0	メモリ・モード・セット					0	0	0	表示モード・セット					コントロール・レジスタ
		W				メモリ・モード・セット								表示モード・セット					
R21	E8002AH	R	0	0	0	0	0	0	テキスト・アクセス・セット							クリアーP.S			
		W							テキスト・アクセス・セット							クリアーP.S			
R22	E8002CH		ソース・ラスター								デスティネーション・ラスター								ラスター・アドレス
R23	E8002EH		ビット・マスク・レジスタ																マスク・データ
---	E80480H	R	0	0	0	0	0	0	0	0	0	0	0	0	ラスターコピー	0	高速クリア	画像入力	CRTC動作ポート
		W													ラスターコピー	0	高速クリア	画像入力	

●表3．8　各画面モードにおける CRTC レジスタ設定値
上段：16進数，下段（）内：10進数

Reg. No（レジスタ・アドレス）	高解像度				低解像度		
	768×512	512×512	512×256	256×256	512×512	512×256	256×256
R00 E80000H	89H (137)	5BH (91)	5BH (91)	2DH (45)	4BH (75)	4BH (75)	25H (37)
R01 E80002H	0EH (14)	09H (9)	09H (9)	04H (4)	03H (3)	03H (3)	01H (-1)
R02 E80004H	1CH (28)	11H (17)	11H (17)	06H (6)	05H (5)	05H (5)	00H (0)
R03 E80006H	7CH (124)	51H (81)	51H (81)	26H (38)	45H (69)	45H (69)	20H (32)
R04 E80008H	237H (567)	237H (567)	237H (567)	237H (567)	103H (259)	103H (259)	103H (259)
R05 E8000AH	005H (5)	005H (5)	005H (5)	005H (5)	02H (2)	02H (2)	02H (2)
R06 E8000CH	028H (40)	028H (40)	028H (40)	028H (40)	010H (16)	010H (16)	010H (16)
R07 E8000EH	228H (552)	228H (552)	228H (552)	228H (552)	100H (256)	100H (256)	100H (256)
R08 E80010H	1BH (27)	1BH (27)	1BH (27)	1BH (27)	2CH (44)		24H (36)

〈注意〉

画像取り込み時，R08（E80010H）の値を次のように変更する

- 512×512 モード ┐
  512×256 モード ┘── 9AH (154)
- 256×256 モード ──── EBH (235)

●図3．8　CRTCレジスタの詳細①

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
R9							ラスター割り込み位置									

割り込みを発生するラスター・アドレスを指定，割り込みさせないときはMFPのGPIP6をマスクにセットする

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
R10 R11							X,Y 方向スクロール									

XまたはY方向の表示開始座標を指定する（テキスト画面）

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
R12 R13							スクリーン0（X，Yの順）									
R14 R19								スクリーン1〜3（X，Yの順）								

XまたはY方向の表示開始座標位置を指定する（グラフィック画面）

R20（上位バイト）

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
			0	0			

- D_9 D_8
  - 0 0 —— グラフィック16色4画面モード
  - 0 1 —— グラフィック256色2画面モード
  - 1 1 —— グラフィック 65,536色1画面モード
  - （D_9＝“1”，D_8＝“0”は無効）
- D_{10}
  - 0 —— グラフィック仮想画面512ドット・モード
  - 1 —— グラフィック仮想画面1024ドット・モード（必ずD_8＝“0”，D_9＝“0”にする）

（下位バイト）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

D_4	D_3	D_2	D_1	D_0			
0	0	0	0	0	低解像度（15.98KHz）	256·256	
0	0	0	0	1	低解像度（15.98KHz）	512·256	
0	0	1	0	1	低解像度（15.98KHz）	512·512	（インターレース）
1	0	0	0	0	高解像度（31.5KHz）	256·256	（ラスター2度読み）
1	0	0	0	1	高解像度（31.5KHz）	512·256	（ラスター2度読み）
1	0	1	0	1	高解像度（31.5KHz）	512·512	
1	0	1	1	0	高解像度（31.5KHz）	768·512	

R21（上位バイト）

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8

- D_8
  - 0 —— テキスト・メモリCPUシングル・アクセス（R21のD_4ーD_7は無効）
  - 1 —— テキスト・メモリCPU同時アクセス
- D_9
  - 0 —— テキストCPUアクセス・ビット・マスクOFF
  - 1 —— テキストCPUアクセス・ビット・マスクON

D_7 D_6 D_5 D_4

(下位バイト)　(R21のD_8が1の時有効)

- D_4: 0 —— テキストT0プレーン同時アクセスOFF
  1 —— テキストT0プレーン同時アクセスON
- D_5: 0 —— テキストT1プレーン同時アクセスOFF
  1 —— テキストT1プレーン同時アクセスON
- D_6: 0 —— テキストT2プレーン同時アクセスOFF
  1 —— テキストT2プレーン同時アクセスON
- D_7: 0 —— テキストT3プレーン同時アクセスOFF
  1 —— テキストT3プレーン同時アクセスON

〈D_0——D_3〉
テキスト・ラスター・コピーの同時アクセス・プレーン・セレクト

1,024 × 1,024

テキストT3プレーン(E60000H—E7FFFEH)〈D_3に対応〉
テキストT2プレーン(E40000H—E5FFFEH)〈D_2に対応〉
テキストT1プレーン(E20000H—E3FFFEH)〈D_1に対応〉
テキストT0プレーン(E00000H—E1FFFEH)〈D_0に対応〉

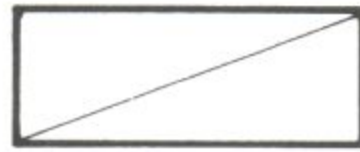
D_3 D_2 D_1 D_0

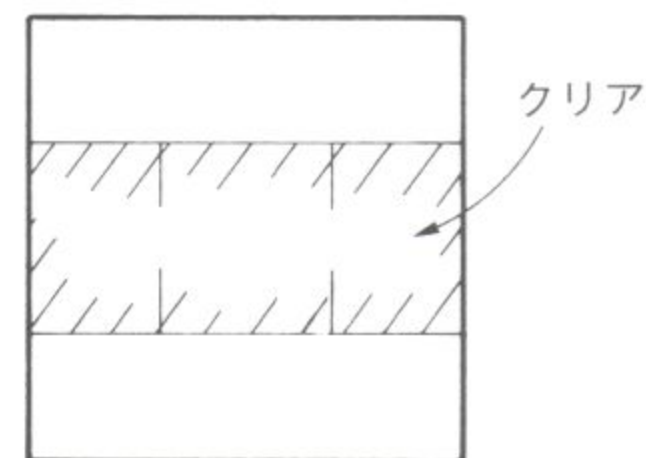
- D_0: 0 —— テキストT0プレーン・ラスター・コピー同時アクセスOFF
  1 —— テキストT0プレーン・ラスター・コピー同時アクセスON
- D_1: 0 —— テキストT1プレーン・ラスター・コピー同時アクセスOFF
  1 —— テキストT1プレーン・ラスター・コピー同時アクセスON
- D_2: 0 —— テキストT2プレーン・ラスター・コピー同時アクセスOFF
  1 —— テキストT2プレーン・ラスター・コピー同時アクセスON
- D_3: 0 —— テキストT3プレーン・ラスター・コピー同時アクセスOFF
  1 —— テキストT3プレーン・ラスター・コピー同時アクセスON

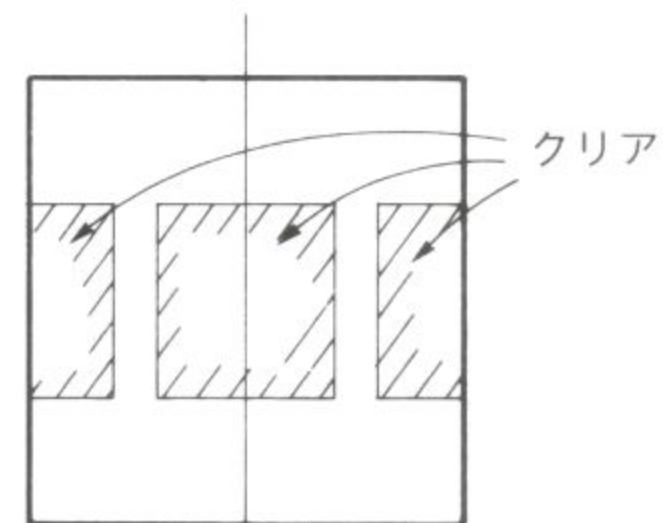
(続く)

●図3．8　CRTC レジスタの詳細②

グラフィック仮想画面1,024×1,024を高速クリアする場合（R20のD_{10}が1のとき有効）

D_3　D_2　D_1　D_0

（D_3～D_0は無効）

［水平768,512ドット・モード の場合］

高速クリアされる領域は，水平方向が，0～1,023ドットすべて，垂直方向が，表示領域の垂直範囲になる．

［水平256ドット・モードの場合］

高速クリアされる領域は，水平方向が，上図の領域、垂直方向が，表示領域の垂直範囲になる．

グラフィック仮想画面512×512を高速クリアする場合（R20のD_{10}が0のとき有効）

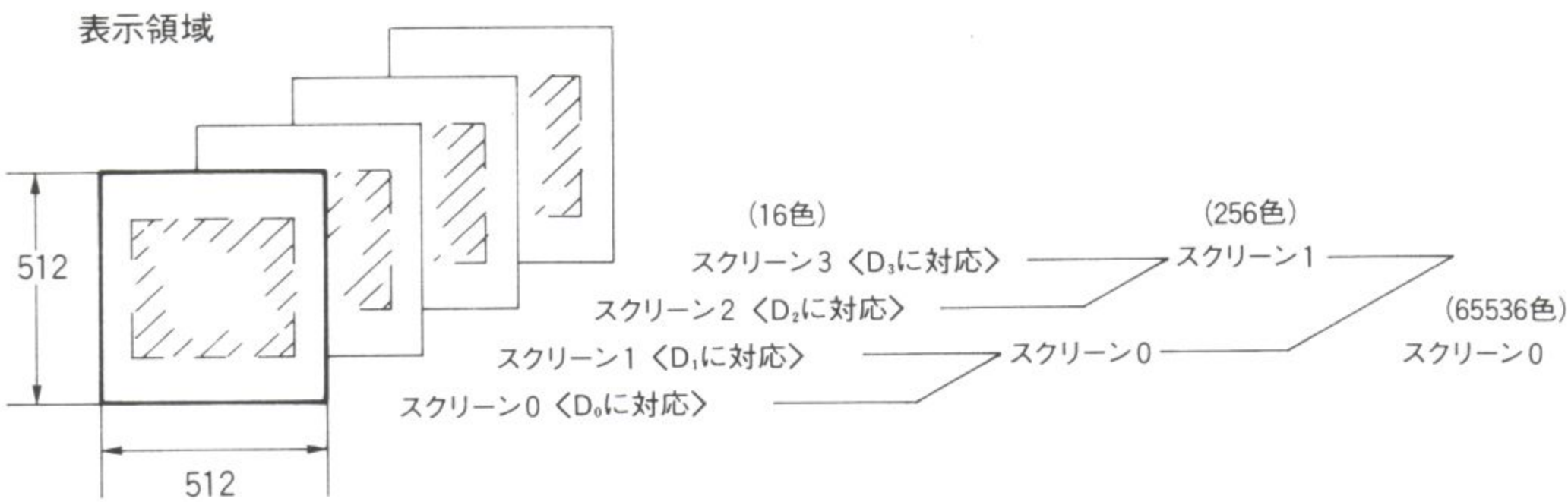
表示領域

512

512

(16色)
スクリーン3〈D_3に対応〉
スクリーン2〈D_2に対応〉
スクリーン1〈D_1に対応〉
スクリーン0〈D_0に対応〉

(256色)
スクリーン1
スクリーン0

(65536色)
スクリーン0

［グラフィック16色4画面モード（R20のD_{10}が0，D_9が0，D_8が0のとき有効）］

D_3　D_2　D_1　D_0

D_0:
0 ── グラフィックスクリーン0表示領域高速クリアOFF
1 ── グラフィックスクリーン0表示領域高速クリアON

D_1:
0 ── グラフィックスクリーン1表示領域高速クリアOFF
1 ── グラフィックスクリーン1表示領域高速クリアON

D_2:
0 ── グラフィックスクリーン2表示領域高速クリアOFF
1 ── グラフィックスクリーン2表示領域高速クリアON

D_3:
0 ── グラフィックスクリーン3表示領域高速クリアOFF
1 ── グラフィックスクリーン3表示領域高速クリアON

［グラフィック256色2画面モード（R20のD_{10}が0，D_9が0，D_8が1のとき有効）］

D_3　D_2　D_1　D_0

D_1 D_0:
0　0 ── グラフィックスクリーン0表示領域高速クリアOFF
1　1 ── グラフィックスクリーン0表示領域高速クリアON

D_3 D_2:
0　0 ── グラフィックスクリーン1表示領域高速クリアOFF
1　1 ── グラフィックスクリーン1表示領域高速クリアON

（続く）

●図3.8　CRTCレジスタの詳細③

［グラフィック65,536色1画面モード（R20のD_{10}が0，D_9が1，D_8が1のとき有効）］

D_3　D_2　D_1　D_0

0　0　0　0 ―― グラフィック・スクリーン0表示領域高速クリアOFF
1　1　1　1 ―― グラフィック・スクリーン0表示領域高速クリアON

なお，256色2画面モードと65536色1画面モードにおいて，上記以外のデータを設定した場合は，上図に該当するビットのスクリーンの表示領域が高速クリアされる．

R 22

D_{15} D_{14} D_{13} D_{12} D_{11} D_{11} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

ソース・ラスター	デスティネーション・ラスター

ラスター転送のソース・ラスター・アドレス　　ラスター転送のデスティネーション・ラスター
R21の$D_{0\sim3}$に同時アクセスを設定し，動作ポートのD_3＝1にするとコピーが行なわれる．

R 23

D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8 D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

ビット・マスク

任意のビットをマスクして演算効率を改善したいとき，マスク・パターンを指定する．
1のときマスクされる．R21のD_9ビットで発効する．

・CRTC動作設定ポート（E80480H）
CRTCが，デュアルポートDRAMのSAM（シリアル・アクセス・メモリ）を介して各機能を動作させる．
また，各機能は，リードとライト操作によって異なる意味をもつ．

・ライト操作

D_7　D_6　D_5　D_4　D_3　D_2　D_1　D_0

（D_2 = 0）

D_0：
0 ―― グラフィックRAMへの画像入力停止（通常表示）
1 ―― グラフィックRAMへの画像入力実行

D_1：
1 ―― グラフィックRAM高速クリア実行

D_3：
1 ―― テキスト・ラスター・コピー実行
0 ―― テキスト・ラスター・コピー中止

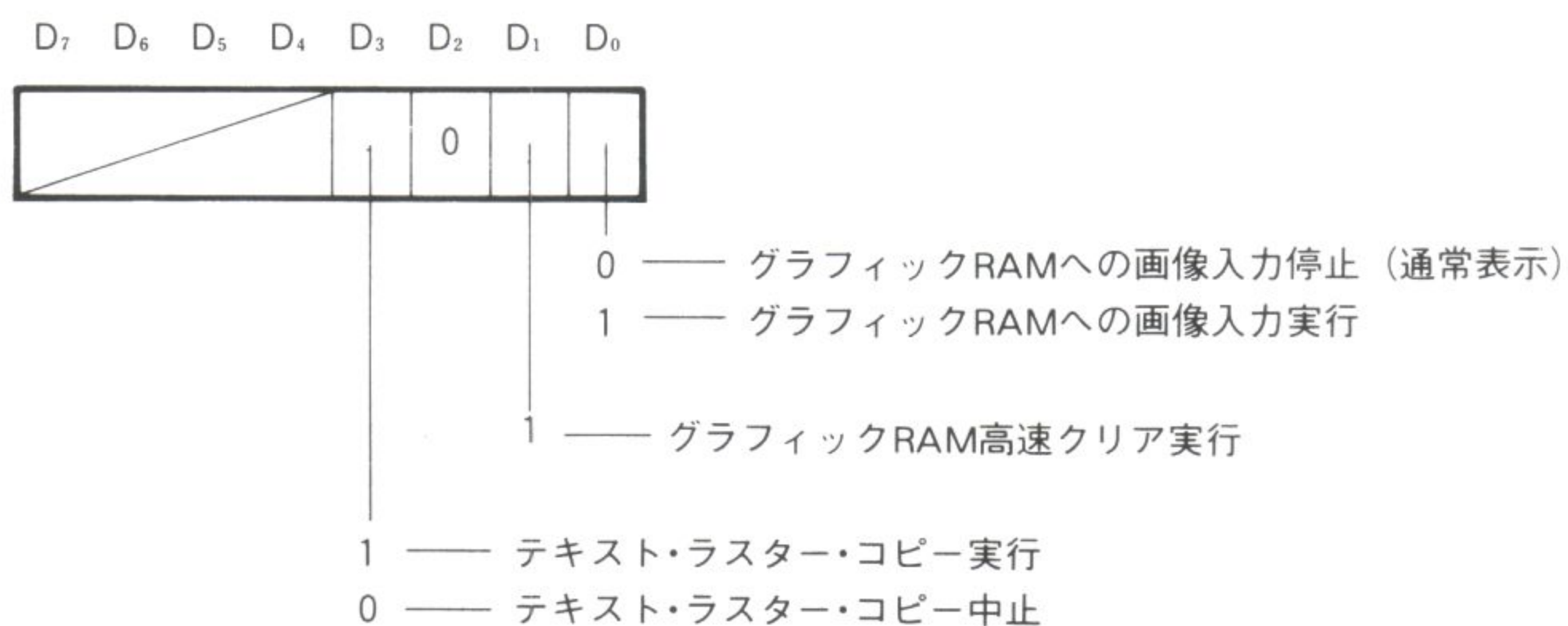
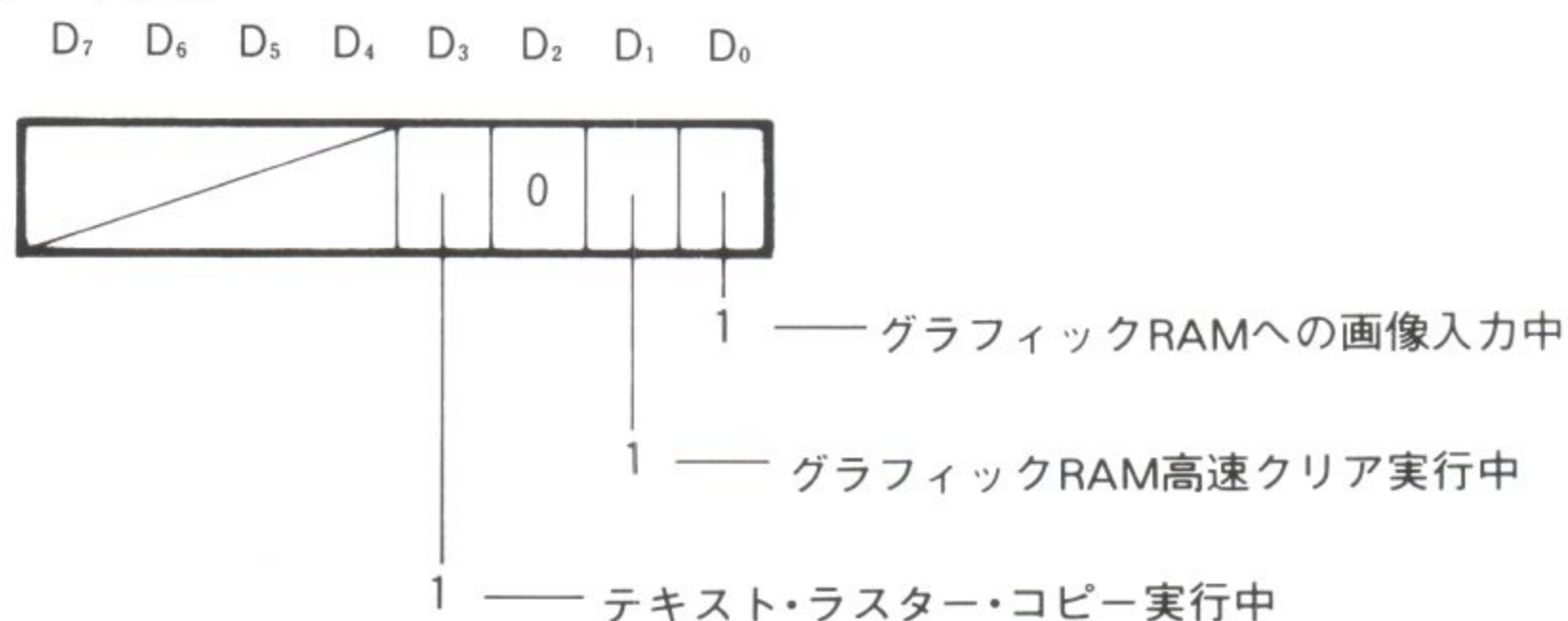
・リード操作

D_7　D_6　D_5　D_4　D_3　D_2　D_1　D_0

（D_2 = 0）

D_0：
1 ―― グラフィックRAMへの画像入力中

D_1：
1 ―― グラフィックRAM高速クリア実行中

D_3：
1 ―― テキスト・ラスター・コピー実行中

＊D_0，D_1がともに0のときのみ，グラフィックは通常表示

3-5 スプライトの制御

X68000の画像処理の特徴は，何と言っても**スプライト**にあります．スプライトの利点は，**重ね合わせ**ができることと，**移動**が簡単にできることです．もしこれがないと，動く画像の表示には一部画面のセーブ，復元を頻繁に繰り返さざるを得なくなり，描画速度がガタ落ちします．

X68000におけるスプライトの仕様は表3.9のとおりで，画面上になんと128個までのスプライトと，**バックグラウンド**(最大2面)が同時に表示できるというスーパーぶりです．これらは上下左右に反転できる機能をもっているため，「裏返し」する際に描画し直す必要もありません．

スプライトは専用のICで制御され，レジスタなどは図3.9のような配置になっています．また，**TEXTエリア**はバックグラウンドを収容し，表3.10のフォーマットになっています．TEXTエリアは，必要に応じてスプライトの拡張エリアとしても使用できます．**PCGエリア**は個々のスプライトを収容する部分で，表3.11のフォーマットで使われます．

レジスタ関係では，スクロール・レジスタ(アドレス・マップは表3.12のとおり)が先頭部分に配置され

●図3．9　スプライト・レジスタ・アドレス・マップ

16bit

EB0000H
EB0002H
SPスクロール・レジスタ
EB0400H
Reserved
EB0800H
BGスクロール・レジスタ
EB080AH
画面モード・レジスタ
EB0812H
Reserved
EB8000H
PCGエリア
EBC000H
TEXTエリア0
EBE000H
TEXTエリア1
EBFFFEH

レジスタ・アドレス

PCGアドレス

PCGエリア

PCGエリアとTEXTエリアは，PCGアドレスの16Kワードを基本的に左図のように2分して割り当てているが，BGの使い方で下図のような場合も可能である．

例．1
EB8000H
PCGエリア
EBC000H
TEXTエリア0
EBE000H
TEXTエリア1
EBFFFEH

例．2
PCGエリア
TEXTエリア1

例．3
PCGエリア

1．TEXTエリア0,1は，BGを2面表示するためのエリアである．BG0,1がそれぞれどちらのTEXTエリアを使用するかはBGスクロール・レジスタ中の“TEXT SEL”ビットで設定できる．

2．BGを1面しか表示しない場合などのように，TEXTエリアを1面しか使用しない場合は，余ったほうのTEXTエリアをPCGエリアとして使用できる．

3．BGをまったく表示しない場合は，16Kワードすべてを PCGエリアとして使用できる．その場合，定義可能なSPパターンは256パターンになる．

●表3．9　スプライト仕様

項目		内容	備考
スプライト	SPパターン定義	サイズ　16×16ドット/パターン 定義数　通常128パターン(BGを表示しない場合最大×256パターン定義可能) 色　1パターンにつき 16色/65,536色（ドット単位） 画面全体で　256色/65,536色	
	SP表示	SP仮想座標系　1,024×1,024 ドット 表示画面　水平：512 ドット or 256 ドット 垂直：512 ライン or 256 ライン 表示制限　128 SP/画面 32 SP/ライン	
	その他の機能	・H反転（左右の入れ換え） ・V反転（上下の入れ換え） ・BGとのプライオリティ （ただし，これらの機能は各SP単位に設定可能）	・プライオリティ (BG0＞BG1) (SP0＞SPn＞SP127)
バックグラウンド	BGパターン定義	サイズ　8×8 ドット/パターン 16×16 ドット/パターン 定義数　8×8 ドット/パターンの場合，max256パターン 16×16 ドット/パターンの場合，通常128パターン 色　1パターンにつき 16色/65,536色（ドット単位） 画面全体で　256色/65,536色	BGパターンとSPパターンは共用
	BG表示	テキスト座標系　max1,024×1,024 ドット 表示画数　max 2面（2面の独立スクロール可能） 表示画面　水平：512 ドット or 256 ドット 垂直：512 ライン or 256 ライン 表示制限　512 ドット表示時はBG1面のみ表示 (BGパターン・サイズは16×16ドットに固定) 256 ドット表示時はBG2面同時表示 (BGパターン・サイズは8×8ドットに固定)	
	その他の機能	・H反転 ・V反転 （ただし，これらの機能は各BG単位に設定可能）	

●表3．10　TEXTエリア

名称		レジスタ・アドレス	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備考
TEXTエリア	TEXTエリア0	EBC000	V反転(BG)	H反転(BG)	0	0	←COLOR(BG)→				←BG Code→								
		EBC002																	
		〜																	
		EBDFFC																	
		EBDFFE	V反転(BG)	H反転(BG)	0	0	←COLOR(BG)→				←BG Code→								
	TEXTエリア1	EBE000	V反転(BG)	H反転(BG)	0	0	←COLOR(BG)→				←BG Code→								
		EBE002																	
		〜																	
		EBFFFC																	
		EBFFFE	V反転(BG)	H反転(BG)	0	0	←COLOR(BG)→				←BG Code→								

●表3．11　PCGエリア

名　称		レジスタ・アドレス	ビット構成 D15 D14 D13 D12	D11 D10 D9 D8	D7 D6 D5 D4	D3 D2 D1 D0	備　考
SP コード0	0	EB8000	I G R B	I G R B	I G R B	I G R B	・各SPコードに対するPCG配置 (16Bit×16Bit: 0 2 / 1 3) ・水平512ドット・モード SPコード・サイズ＝上図(0123) ＝BGコード・サイズ ＝16×16ドット ・水平256ドット・モード SPコード・サイズ＝上図(0123) ＝16×16ドット BGコード・サイズ上図(0) または 上図(1) または 上図(2) または 上図(3) ＝8×8ドット
			0	1	2	3	
		EB8002	I G R B	I G R B	I G R B	I G R B	
			4	5	6	7	
		〜					
		EB801C	I G R B	I G R B	I G R B	I G R B	
			56	57	58	59	
		EB801E	I G R B	I G R B	I G R B	I G R B	
			60	61	62	63	
	1	EB8020	I G R B	I G R B	I G R B	I G R B	
			64	65	66	67	
		〜					
		EB803E	I G R B	I G R B	I G R B	I G R B	
			124	125	126	127	
	2	EB8040	I G R B	I G R B	I G R B	I G R B	
			128	129	130	131	
		〜					
		EB805E	I G R B	I G R B	I G R B	I G R B	
			188	189	190	191	
	3	EB8060	I G R B	I G R B	I G R B	I G R B	
			192	193	194	195	
		〜					
		EB807E	I G R B	I G R B	I G R B	I G R B	
			252	253	254	255	
SP コード1		EB8080	I G R B	I G R B	I G R B	I G R B	
			0	1	2	3	
		〜					
		EB80FE	I G R B	I G R B	I G R B	I G R B	
			252	253	254	255	
SP コード2		EB8100	I G R B	I G R B	I G R B	I G R B	
			0	1	2	3	
		〜					
		EB817E	I G R B	I G R B	I G R B	I G R B	
			252	253	254	255	
SP コード127		EBBF80	I G R B	I G R B	I G R B	I G R B	
			0	1	2	3	
		〜					
		EBBFFE	I G R B	I G R B	I G R B	I G R B	
			252	253	254	255	

＊定義できるSPコードおよびBGコードの最大値は、PCGエリアの大きさによって決定.
1．TEXTエリア＝EBC000H 〜 EBFFFEH（8Kワード）の場合，コードは0〜127
2．TEXTエリア＝EBE000H 〜 EBFFFEH（4Kワード）の場合，コードは0〜191
3．BGを表示しない場合，SPコードは0〜255

●表3．12　スプライト・レジスタ・アドレス・マップ（全レジスタREAD/WRITE 可）
［スプライト，バックグラウンド・スクロール・レジスタ］

名　称		レジスタ・アドレス	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備　考
SPスクロール・レジスタ	SP 0	EB0000	0	0	0	0	0	0	X座標										1個のスプライトについて左の4ワードのレジスタが割り当てられる
		EB0002	0	0	0	0	0	0	Y座標										
		EB0004	V反転(SP)	H反転(SP)	0	0	COLOR (SP)				SP　CODE								
		EB0006	0	0	0	0	0	0	0	0	0	0	0	0	0	0 拡張用	PRW		
		∫	∫																
	SP 127	EB03F8	0	0	0	0	0	0	X座標										
		EB03FA	0	0	0	0	0	0	Y座標										
		EB03FC	V反転(SP)	H反転(SP)	0	0	COLOR (SP)				SP　CODE								
		EB03FE	0	0	0	0	0	0	0	0	0	0	0	0	0	0 拡張用	PRW		
BGスクロール・レジスタ	BG 0	EB0800	0	0	0	0	0	0	X座標										
		EB0802	0	0	0	0	0	0	Y座標										
	BG1	EB0804	0	0	0	0	0	0	X座標										
		EB0806	0	0	0	0	0	0	Y座標										
	BGコントロール	EB0808	0	0	0	0	0	0 拡張用	DISP / CPU	0	0	0	0 TEXTSEL (BG1)		BG1 ON / OFF	0 TEXTSEL (BG0)		BG0 ON / OFF	
画面モード・レジスタ		EB080A	0	0	0	0	0	0	0	0	H　total								水平トータル
		EB080C	0	0	0	0	0	0	0	0	0	0	H　disp						水平表示開始位置
		EB080E	0	0	0	0	0	0	0	0	V　disp								垂直表示開始位置
		EB0810	0	0	0	0	0	0	0	0	0	0	0	L/H freq	0 V Res.		0 H Res.		解像度

●図3．10　スクロール・レジスタの設定内容①

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
＋0	0	0	0	0	0	0	SPスクロールX座標									
＋2	0	0	0	0	0	0	SPスクロールY座標									

スプライト(SP)の表示開始位置を仮想座標で指定

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
＋4	V反転(SP)	H反転(SP)	0	0	カラー・コード				SPコード							

- D_{15}：1で上下反転
- D_{14}：1で左右反転
- カラー・コード：パレット・アドレスの上位4ビット(表のCOLOR)
- SPコード：PCGエリアの中で表示するパターンの番号

パレット・アドレス COLOR D_{11} D_{10} D_9 D_8	パレット・アドレス PCGデータ I G R B	レジスタ・アドレス	D_{15}—D_{11} Green	D_{10}—D_6 Red	D_5—D_1 Blue	D_0 I	備考
0 0 0 0 (Table 0)	0 0 0 0	E82200H	←16Bit→				(透明) テキスト・パレットと共通
	0 0 0 1	E82202H					
	:	:					
	1 1 1 1	E8221EH					
0 0 0 1 (Table 1)	0 0 0 0	E82220H					(透明)
	0 0 0 1	E82222H					
	:	:					
	1 1 1 1	E8223EH					
	:	:					
1 1 1 1 (Table15)	0 0 0 0	E823E0H					(透明)
	0 0 0 1	E823E2H					
	:	:					
	1 1 1 1	E823FEH					

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
＋6	0	0	0	0	0	0	0	0	0	0	0	0	0	0 拡張用	PRW	

PRW：表示優先順位

(続く)

●図3．10　スクロール・レジスタの設定内容②

D_1	D_0	表示優先順位
0	0	スプライトを表示しない
0	1	BG 0 > BG 1 > SP
1	0	BG 0 > SP > BG 1
1	1	SP > BG 0 > BG 1

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9–D_0
+800	0	0	0	0	0	0	BG 0 スクロールX座標
+802	0	0	0	0	0	0	BG 0 スクロールY座標

BG 0 の表示開始位置を仮想座標で指定

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9–D_0
+804	0	0	0	0	0	0	BG 1 スクロールX座標
+806	0	0	0	0	0	0	BG 1 スクロールY座標

BG 1 の表示開始位置を仮想座標で指定

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
+808	0	0	0	0	0	0 拡張用	DISP /CPU	0	0	0	TEXT SEL (BG 1)		BG 1 ON/OFF	TEXT SEL (BG 0)		BG 0 ON/OFF

DISP/CPU　0：CPUアクセスのため表示OFF
　　　　　1：表示ON．CPUアクセスは遅くなる

TEXT SEL　0：エリア0
　　　　　1：エリア1

BG 1 (0)　0：OFF
　　　　　1：ON

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
+80A	0	0	0	0	0	0	0	0	水平トータル							
+80C	0	0	0	0	0	0	0	0	0	0	水平表示開始位置					
+80E	0	0	0	0	0	0	0	0	垂直表示開始位置							

これらのレジスタはCRTCと同じ値をセットする

	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
+810	0	0	0	0	0	0	0	0	0	0	0	L/H 周波数	垂直解像度		水平解像度	

L/H周波数 0 ＝低解像度(15.98KHz)
　　　　　1 ＝高解像度(31.5KHz)

垂直解像度
(表(a)のとおり)

水平解像度
(表(b)のとおり)

(a)

V Res.	00	01	10	11
垂直解像度	256ライン	512ライン	不　可	不　可

(b)

H Res.	00	01	10	11
水平解像度	256ドット	512ドット	不　可	不　可

ています．これらの個々の設定の仕方は，図3.10に示すとおりです．

続く画面モード・レジスタは，表3.13のような設定の仕方をすることになっています．注意点としては，低解像度256×256ドット時にはＨ disp 設定後，130μ 秒待ってＨ total を設定しなければなりません．このための方法の１つとして，たとえば次のような遅延ルーチンを利用します．

●表３．13　画面モード・レジスター覧表
上段：16進数，下段（　）内：10進数

画面モード・レジスタ	高解像度			低解像度		
	512×512	512×256 2度読み 2ライン/ドット	256×256 2度読み	512×512 インターレース	512×256	256×256
H total (EB080AH)	FFH (255)	FFH (255)	FFH (255)	FFH (255)	FFH (255)	25H (37)
H disp (EB080CH)	15H (21)	15H (21)	0AH (10)	09H (9)	09H (9)	04H (4)
V disp (EB080EH)	28H (40)	28H (40)	28H (40)	10H (16)	10H (16)	10H (16)
L/H freq / V Res. / H Res. (EB0810H)	15H (21)	11H (17)	10H (16)	05H (5)	01H (1)	00H (0)
BG表示	1面	1面	2面	1面	1面	2面

●図３．11　TEXT エリアのアドレス配置

(EB0000H＋××××H)
←64パターン→

C000	C002	C004	C006	……	C07C	C07E
C080	C082	C084	C086	……	C0FC	C0FE
DF00	DF02	DF04	DF06	……	DF7C	DF7E
DF80	DF82	DF84	DF86	……	DFFC	DFFE

BG：８×８ドット/パターン構成の場合
512ドット
512ドット

BG：16×16ドット/パターン構成の場合
1,024ドット
1,024ドット

(EB0000H＋××××H)
←64パターン→

E000	E002	E004	E006	……	E07C	E07E
E080	E082	E084	E086	……	E0FC	E0FE
FF00	FF02	FF04	FF06	……	FF7C	FF7E
FF80	FF82	FF84	FF86	……	FFFC	FFFE

BG：８×８ドット/パターン構成の場合
512ドット
512ドット

BG：16×16ドット/パターン構成の場合
1,024ドット
1,024ドット

```
delay :
        move.b  #$FF, d0
dloop :
        dbra    d0, dloop
        rts
```

スプライト関係のエリアと図形との対応は，TEXT エリアについては図3.11，PCG エリアについては図3.12に示します．

●図3．12　PCG エリアのアドレス配置

16ドット

16ドット

0

SPコード＝0
BGコード＝0

BG：16×16ドット/パターン構成

SPコード＝BGコード＝n
(n＝0，1……)

16ドット

16ドット

0	2
1	3

SPコード＝0
BGコード＝0，1，2，3

BG：8×8ドット/パターン構成

SPコード＝n
BGコード＝4n，4n＋1，4n＋2，4n＋3

・PCGアドレスとPCGコードの対応は以下のとおり．

PCGアドレス	16Bitデータ
EB8000H	
EB8020H	
EB8040H	
EB8060H	
EB8080H	
EB80A0H	
EB80C0H	
EB80E0H	
EB8100H	
EB8120H	
EB8140H	
EB8160H	
EB8180H	
EB81A0H	
EB81C0H	
EB81E0H	
⋮	⋮

BG：16×16ドット/パターンの場合

SPコード	BGコード
0	0
1	1
2	2
3	3
⋮	⋮

＋80H

BG：8×8ドット/パターンの場合

SPコード	BGコード
0	0
	1
	2
	3
1	4
	5
	6
	7
2	8
	9
	10
	11
3	12
	13
	14
	15
⋮	⋮

＋20H

3-6 ビデオ・コントローラの設定

ビデオ・コントローラの役割りは，基本的には表示データをドット単位に送出するところにありますが，X68000の場合，**複数画面**を**合成**することが最重点機能になっている感じがあります．このことは，アドレス・マップ(表3.14)を見れば明らかで，画面のON・OFFやプライオリティなどの項目で満たされているようすがわかります．

●表3．14　ビデオ・コントローラ・レジスタ・アドレス・マップ

<table>
<tr><th>レジスタ・アドレス</th><th>D_{15}</th><th>D_{14}</th><th>D_{13}</th><th>D_{12}</th><th>D_{11}</th><th>D_{10}</th><th>D_9</th><th>D_8</th><th>D_7</th><th>D_6</th><th>D_5</th><th>D_4</th><th>D_3</th><th>D_2</th><th>D_1</th><th>D_0</th></tr>
<tr><td rowspan="3">Reg. 1
E82400H</td><td colspan="13">リ　ザ　ー　ブ</td><td colspan="3">メモリ・モード設定</td></tr>
<tr><td colspan="13" rowspan="2"></td><td>G.サイズ</td><td colspan="2">G. モード</td></tr>
<tr><td>1,024/512</td><td>CL1</td><td>CL0</td></tr>
<tr><td rowspan="3">Reg. 2
E82500H</td><td colspan="2">リザーブ</td><td colspan="6">Gr./Tx./Sp. 間プライオリティ設定</td><td colspan="8">グラフィック間，面プライオリティ設定</td></tr>
<tr><td colspan="2" rowspan="2"></td><td colspan="2">スプライト</td><td colspan="2">テキスト</td><td colspan="2">グラフィック</td><td colspan="2">プライオリティ4</td><td colspan="2">プライオリティ3</td><td colspan="2">プライオリティ2</td><td colspan="2">プライオリティ1</td></tr>
<tr><td>SP1</td><td>SP0</td><td>TX1</td><td>TX0</td><td>GR1</td><td>GR0</td><td colspan="2">スクリーンNo.</td><td colspan="2">スクリーンNo.</td><td colspan="2">スクリーンNo.</td><td colspan="2">スクリーンNo.</td></tr>
<tr><td rowspan="2">Reg. 3
E82600H</td><td colspan="8">特　殊　モ　ー　ド</td><td rowspan="2"></td><td>スプライト表示ON/OFF</td><td>テキスト表示ON/OFF</td><td>1,024×1,024グラフィック表示ON/OFF</td><td colspan="4">512×512グラフィック表示ON/OFF</td></tr>
<tr><td>Ys</td><td>AH</td><td>@@VHT</td><td>EXON</td><td>H/P</td><td>B/P</td><td>G/G</td><td>G/T</td><td>SON</td><td>TON</td><td>GS4</td><td>GS3</td><td>GS2</td><td>GS1</td><td>GS0</td></tr>
</table>

・全レジスタともREAD/WRITE可
・@@のVHT信号は，オプションの“デジタイズテロッパ”（仮称）にて使用

●図3．13　ビデオ・コントローラの各レジスタの設定の仕方①

●グラフィック・メモリ仮想画面サイズと色モードの設定（CRTCのR20のD_{10}-D_8と同値を設定）

E82400H　D_2　D_1　D_0

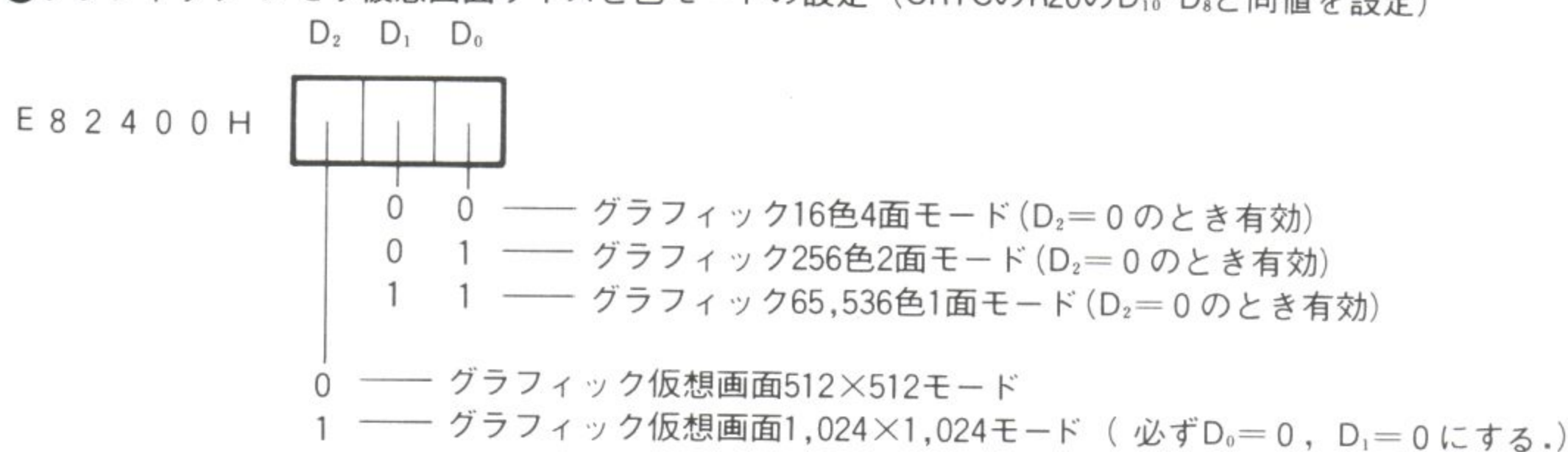
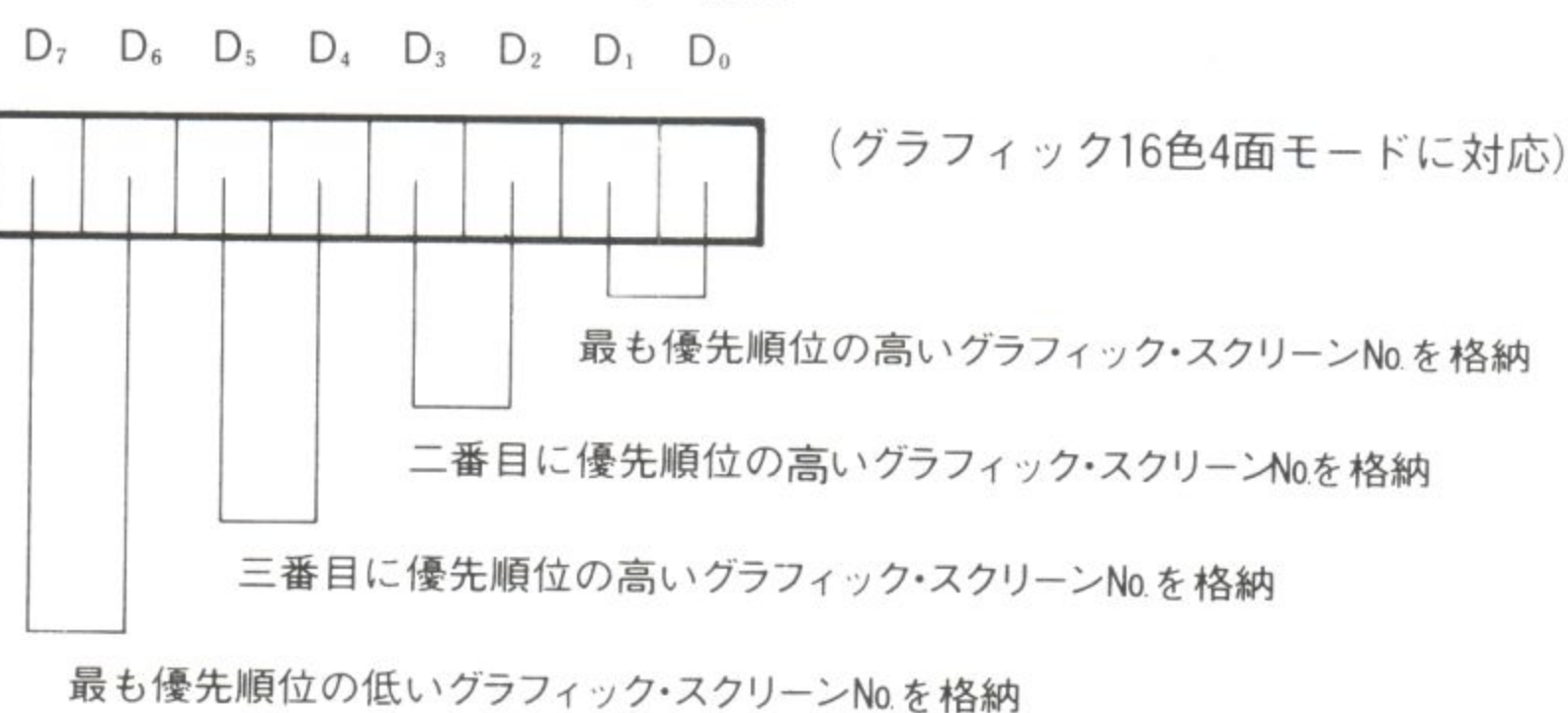
0　0 —— グラフィック16色4面モード(D_2＝0のとき有効)
0　1 —— グラフィック256色2面モード(D_2＝0のとき有効)
1　1 —— グラフィック65,536色1面モード(D_2＝0のとき有効)

0 —— グラフィック仮想画面512×512モード
1 —— グラフィック仮想画面1,024×1,024モード（必ずD_0＝0，D_1＝0にする．）

●グラフィック間の面プライオリティの設定

D_7　D_6　D_5　D_4　D_3　D_2　D_1　D_0

（グラフィック16色4面モードに対応）

最も優先順位の高いグラフィック・スクリーンNo.を格納
二番目に優先順位の高いグラフィック・スクリーンNo.を格納
三番目に優先順位の高いグラフィック・スクリーンNo.を格納
最も優先順位の低いグラフィック・スクリーンNo.を格納

ビット1	ビット2	
0	0	グラフィック・スクリーンNo. 0
0	1	グラフィック・スクリーンNo. 1
1	0	グラフィック・スクリーンNo. 2
1	1	グラフィック・スクリーンNo. 3

各レジスタの設定の仕方は図3.13に示しているとおりですが，ここでいう**優先度**は，半透明処理の際に「捨てられない」度合いを示しています．すなわち，優先度の低い画面の部分は，対応位置の高優先画面が透明でない限りビデオ・コントローラを通過できず，スクリーンで見えなくなります．言い換えると，高優先画面ほど前方にあるように見えます．

一方，**半透明**というのは，合成する2画面間において，高優先画面側で指定された領域の読み出し結果を50：50で重ね合わせることを言います．合成機能は専用TV側でももっており(**スーパー・インポーズ**)，

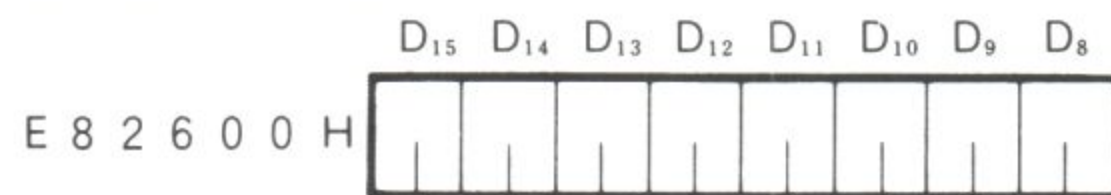
ビット1	ビット0	
0	0	優先順位1
0	1	優先順位2
1	0	優先順位3
1	1	禁止

グラフィック、テキスト、スプライトの設定データに同じ値は設定できない。

(続く)

●図3．13　ビデオ・コントローラの各レジスタの設定の仕方②

●半透明モードと特殊プライオリティ機能の設定

E82600H　D_{15} D_{14} D_{13} D_{12} D_{11} D_{10} D_9 D_8

D_8
0 ── 通常モード
1 ── グラフィック画面の中の任意に指定された領域とテキスト(スプライト)画面との半透明を行なう．ただし，そのグラフィック画面の優先順位が最も高い場合に有効であるほか，グラフィック画面で使用できる色の数は半分になるが，テキスト(スプライト)画面は通常モードで使用できる．

D_9
0 ── 通常モード
ただし，D_{12}＝1，D_{11}＝1のとき使用できるパレットは半分になる．
1 ── D_{12}＝1，D_{11}＝1で，グラフィック画面を2面以上もてるモードにおいて，グラフィック画面間(最も優先順位の高い画面と次に優先順位の高い画面)の指定領域の半透明を行なう．

D_{10}
0 ── Reserve
1 ── D_{12}＝1のとき，半透明，特殊プライオリティを行なう領域指定をグラフィックVRAMのメモリ・データにて行なう．

D_{11}
0 ── D_{12}＝1のとき，特殊プライオリティ・モードとする．
1 ── D_{12}＝1のとき，半透明モードとする．

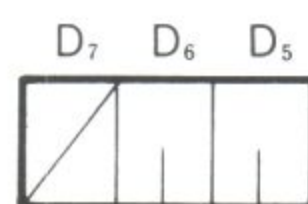
D_{12}
0 ── 通常モード
1 ── 半透明，特殊プライオリティ有効モード

D_{13}
0 ── 通常モード
1 ── 最も優先順位の高いグラフィック画面で指定された領域とテレビ，ビデオ画面との半透明を行なう．

D_{14}
0 ── 通常モード
1 ── テキスト（スプライト）のパレット・メモリ00Hの内容とグラフィック画面との半透明を行なう．

D_{15}
0 ── CMPCUT(Ys) 信号を有効にする．
1 ── CMPCUT(Ys) 信号を強制的にHとし，スーパーインポーズ時でもコンピュータ画面のみを表示する．(TV画面をカットする．)

・テキスト，スプライト表示ON/OFFの設定

D_7 D_6 D_5

D_5
0 ── テキスト画面の表示OFF
1 ── テキスト画面の表示ON

D_6
0 ── スプライト画面の表示OFF
1 ── スプライト画面の表示ON

●グラフィック仮想画面1,024×1,024表示ON/OFFの設定（Reg.1のD_2＝1のとき有効）

D_4

0 —— 画面の表示OFF
1 —— 画面の表示ON

●グラフィック仮想画面512×512表示ON/OFFの設定（Reg.1のD_2＝0のとき有効）

D_3 D_2 D_1 D_0

（グラフィック16色4面モードに対応）

D_0:
0 —— 1番優先順位が高い画面の表示OFF
1 —— 1番優先順位が高い画面の表示ON

D_1:
0 —— 2番目に優先順位が高い画面の表示OFF
1 —— 2番目に優先順位が高い画面の表示ON

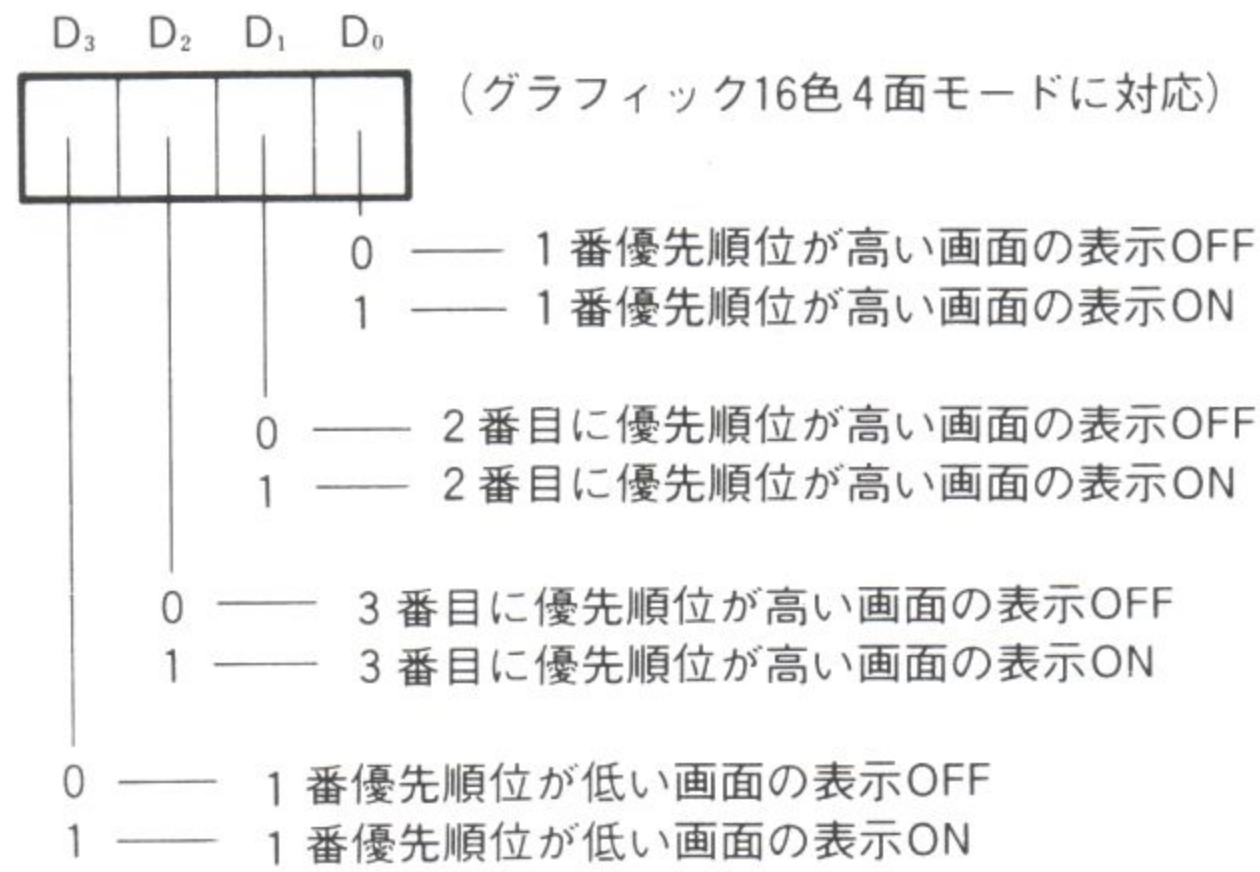
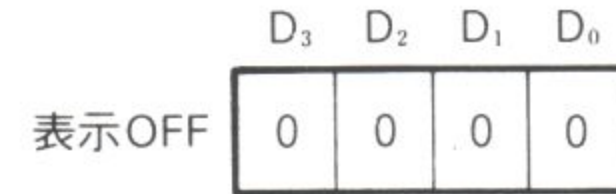
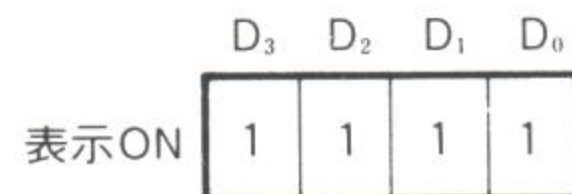
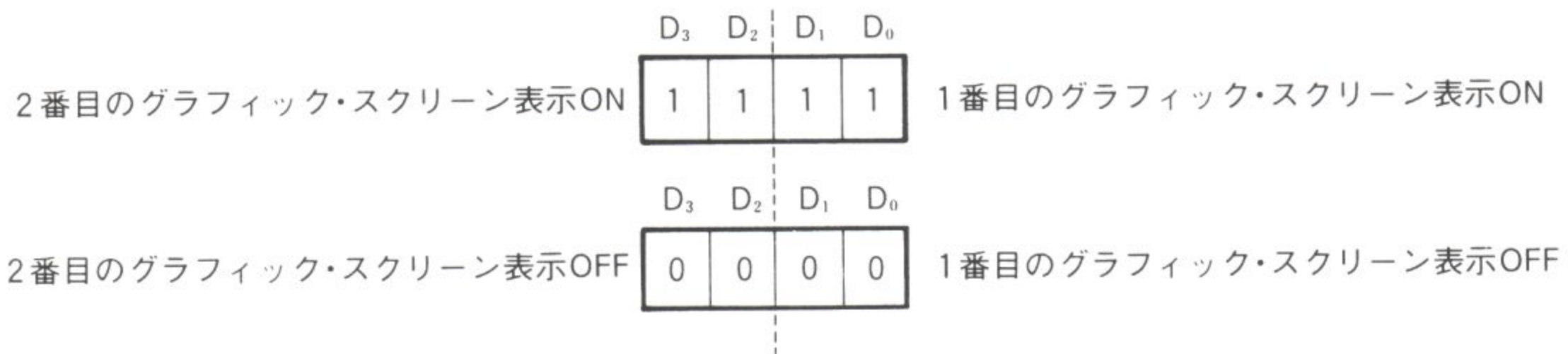
D_2:
0 —— 3番目に優先順位が高い画面の表示OFF
1 —— 3番目に優先順位が高い画面の表示ON

D_3:
0 —— 1番優先順位が低い画面の表示OFF
1 —— 1番優先順位が低い画面の表示ON

・グラフィック65,536色1面モードの場合

	D_3	D_2	D_1	D_0
表示ON	1	1	1	1
表示OFF	0	0	0	0

・グラフィック256色2面モードの場合

	D_3	D_2	D_1	D_0	
2番目のグラフィック・スクリーン表示ON	1	1	1	1	1番目のグラフィック・スクリーン表示ON
2番目のグラフィック・スクリーン表示OFF	0	0	0	0	1番目のグラフィック・スクリーン表示OFF

これと重複させることもできます.

半透明処理は図3.14のように, 同一色データ値を足して２で割る方法で, 半透明指定側のＩ(輝度)信号は捨てられます. また, グラフィックＶRAMで半透明領域の指定を行なうときは, 図3.15のように１つのカラー・プレーンを占有することになります. このことは, 色数が半分に減ってしまうことを意味するので, とくに注意する必要があります. この場合, パレットを参照するときの対応アドレスは０となりますが, 半透明指定側は偶数パレットを参照し, 半透明処理される側の両面は奇数パレットを参照するように設計されているので, 両パレットの内容は一致させておかなければなりません.

●図３．14　半透明処理

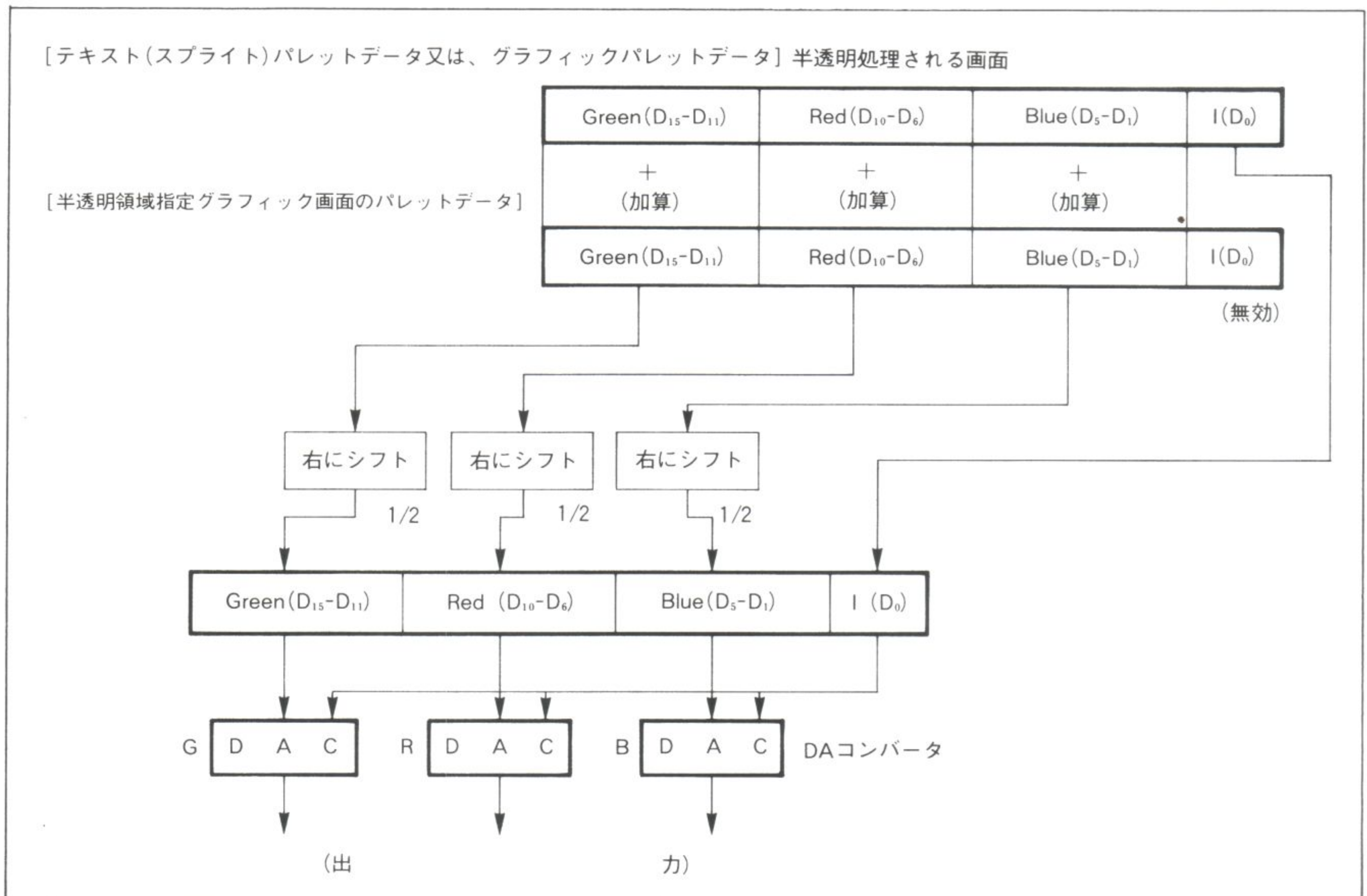



図3.16は，半透明も含めた特殊モードの画面合成例を示したものです．これらは，E826000H の半透明特殊プライオリティ・レジスタの設定の際に参考にしてください．

●図3．15　グラフィックVRAMのメモリ・データ（LSB）を領域指定に使用する場合（D_{10}＝“1”の場合）

スプライト
テキスト
グラフィック・スクリーン1
グラフィック・スクリーン0
G03プレーン
G02プレーン
G01プレーン
G00プレーン
グラフィック・パレット・アドレス
G03 G02 G01 G00
領域指定ビット
パレット・アドレス
D_{15} —— D_0
00H
02H
04H
06H
08H
0AH
0CH
0EH
点線円が，半透明，特殊プライオリティ有効領域

・半透明，特殊プライオリティ機能の領域指定は，表示画面中，最も優先順位の高いグラフィック画面内の任意の部分（ドット単位）に設定可能．
・グラフィックVRAMのメモリ・データのLSBを使用するため，使用できるパレットの数は半分になる．（パレット・アドレスは，偶数アドレスとなる．）

●図3．16　特殊モードの動作例

(a)　CMPCUT(Ys)信号（低解像度スーパーインポーズ時）

TV原画面　＋　コンピュータ画面

Ys(D_{15})＝1のとき

スーパーインポーズ画面

Ys(D_{15})＝0のとき

(b)　半透明モード（×は，任意）

①

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	1	×	×	×	×	×	×

テキスト（スプライト）　＋　グラフィック

②

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	0	1	1	1	0	1

（アミ部分は半透明指定域）

テキスト（スプライト）　＋　グラフィック（高優先）　→　合成画面

⑤

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	1	1	1	1	0	1

（②＋スーパーインポーズ）

＋　TV原画面　→　スーパーインポーズ画面

③

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	0	1	1	1	0	1

グラフィック２画面間での半透明処理
（最優先画面で半透明領域を指定）

④

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	0	1	1	1	1	1

（②＋③）

⑥

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	1	1	1	1	1	0

（③＋スーパーインポーズ）

⑦

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	1	1	1	1	1	1

（④＋スーパーインポーズ）

②～④でD_8，D_9ともに０のとき半透明処理は行なわれないので，領域指定中は０にしておき，必要な段階で図のようにする．

(c) 特殊プライオリティ・モード

①

D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8
×	0	×	1	0	1	×	×

指定領域の優先順位を最高にする

点線内が
特殊プライオリティ
領域

グラフィック・スクリーン1

グラフィック・スクリーン0

テキスト(スプライト)

3-7 キャラクタ・ゼネレータ

テキスト画面に表示される文字データは，**キャラクタ・ゼネレータ**(ROM)によってドットに展開され，テキスト領域に記憶されます．この間の処理は，DOS コールなどのソフトウェアにより対応されるため，通常私たちがこのことを意識しなければならないことはありません．しかし，テキスト画面以外に文字を転送するなど特殊な処理を行なう場合は，プログラム中にそのための機能を含めなければなりません．

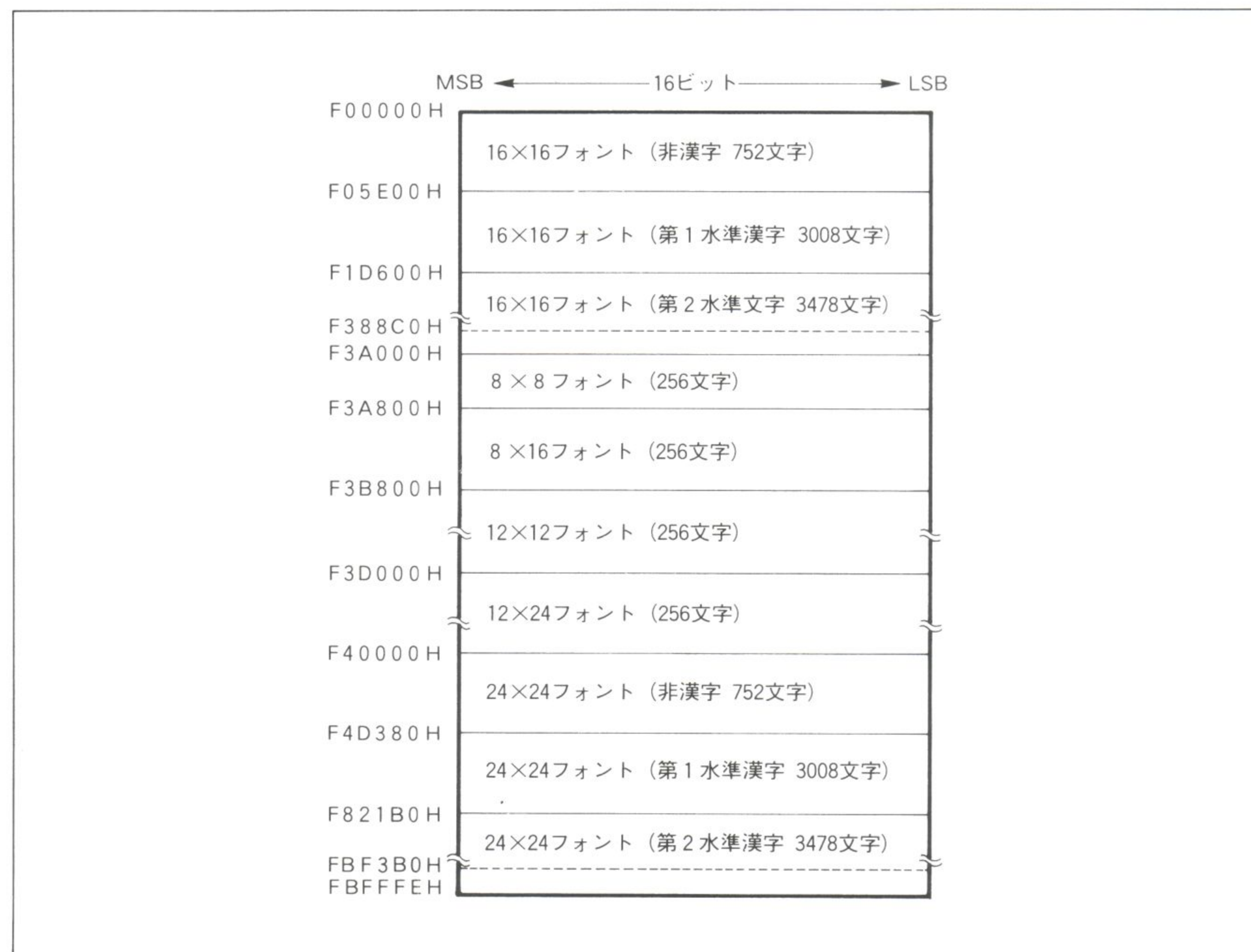
X68000では，表3.15のように，**1/4角**，**半角**，**全角**のそれぞれについて2通りのドット数のフォントをもっています．普通ドット数をいうときは，表のフォント構成を指していますが，実際は文字レター・フェ

●表3．15　CGROM 仕様

文字種	フォント構成	文字レター・フェース	文字数
1/4角文字	8×8，12×12	6×7，10×10	256 文字
半角文字	8×16，12×24	7×13，10×18	256 文字
全角文字	16×16，24×24	15×16，24×24	非漢字――752文字 (JISコード［上位アドレス］21H～28H, ［下位アドレス］21H～7EH) 第1水準漢字――3,008文字 (JISコード［上位アドレス］30H～4FH, ［下位アドレス］21H～7EH) 第2水準漢字――3,478文字 (JISコード［上位アドレス］50H～74H, ［下位アドレス］21H～7EH)

●図3．17　CGROM のアドレス・マップ

MSB ←―― 16ビット ――→ LSB

アドレス	内容
F00000H	16×16フォント（非漢字 752文字）
F05E00H	16×16フォント（第1水準漢字 3008文字）
F1D600H	16×16フォント（第2水準文字 3478文字）
F388C0H	
F3A000H	8×8フォント（256文字）
F3A800H	8×16フォント（256文字）
F3B800H	12×12フォント（256文字）
F3D000H	12×24フォント（256文字）
F40000H	24×24フォント（非漢字 752文字）
F4D380H	24×24フォント（第1水準漢字 3008文字）
F821B0H	24×24フォント（第2水準漢字 3478文字）
FBF3B0H	
FBFFFEH	

ースに示すドット数しか使われておらず，残りは０が書かれています．

これらのCG(キャラクタ・ゼネレータ)ROMは，図3.17のアドレス帯に割り当てられています．フォントごとのアドレス対応は図3.18のとおりで，フォントの構成により１字に対するアドレス帯のサイズが異

●図３．18　CGROMアドレス構成①

●図3．18　CGROMアドレス構成②

なります．漢字コードは字数が多く本書では紹介できませんが，ASCII コード(カナを含む)対文字の表を表3.16に掲げておきます．

●表3．16　アスキーキャラクタ表——JIS 第1水準・第2水準漢字（JIS C 6226-1983に準拠）

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		SH	SX	EX	ET	EQ	AK	BL	BS	HT	LF	VT	FF	CR	SO	SI
1	DE	DC1	DC2	DC3	DC4	NK	SN	EB	CN	EM	SB	EC	→	←	↑	↓
2		!	"	#	$	%	&	'	(	)	*	+	,	—	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	¥	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	￣	·
8		~	¦				を	ぁ	ぃ	ぅ	ぇ	ぉ	ゃ	ゅ	ょ	っ
9	ー	あ	い	う	え	お	か	き	く	け	こ	さ	し	す	せ	そ
A		。	「	」	、	・	ヲ	ァ	ィ	ゥ	ェ	ォ	ャ	ュ	ョ	ッ
B	ー	ア	イ	ウ	エ	オ	カ	キ	ク	ケ	コ	サ	シ	ス	セ	ソ
C	タ	チ	ツ	テ	ト	ナ	ニ	ヌ	ネ	ノ	ハ	ヒ	フ	ヘ	ホ	マ
D	ミ	ム	メ	モ	ヤ	ユ	ヨ	ラ	リ	ル	レ	ロ	ワ	ン	゛	゜
E	た	ち	つ	て	と	な	に	ぬ	ね	の	は	ひ	ふ	へ	ほ	ま
F	み	む	め	も	や	ゆ	よ	ら	り	る	れ	ろ	わ	ん		

第4章

周辺デバイスの制御

4-1 キーボードの制御

キーボード内部には，専用プロセッサとして**80C51**が内蔵されています．そして，本体との間には図4.1のような信号線の接続関係があり，本体側ではMFPが窓口となって，CPUとの間を仲介します．キーボードは，本来のキー入力データの転送以外にも，リモートTVコントロールやマウス制御の仲介機能をもっていますが，それらについては節を改めて紹介します．

80C51との間の転送フォーマットは，表4.1のような調歩式になっており，本体側からはコマンドが，キーボード側からは入力キー対応のコードが図4.2のビット構成で転送されます．

●表4．1　キー・データ転送フォーマット

キー・データ転送手順（シリアル送受信）非同期通信［パリティなし］	項目	値
	ボーレート	2,400ボー
	スタート・ビット	1
	データ長	8
	ストップ・ビット	1

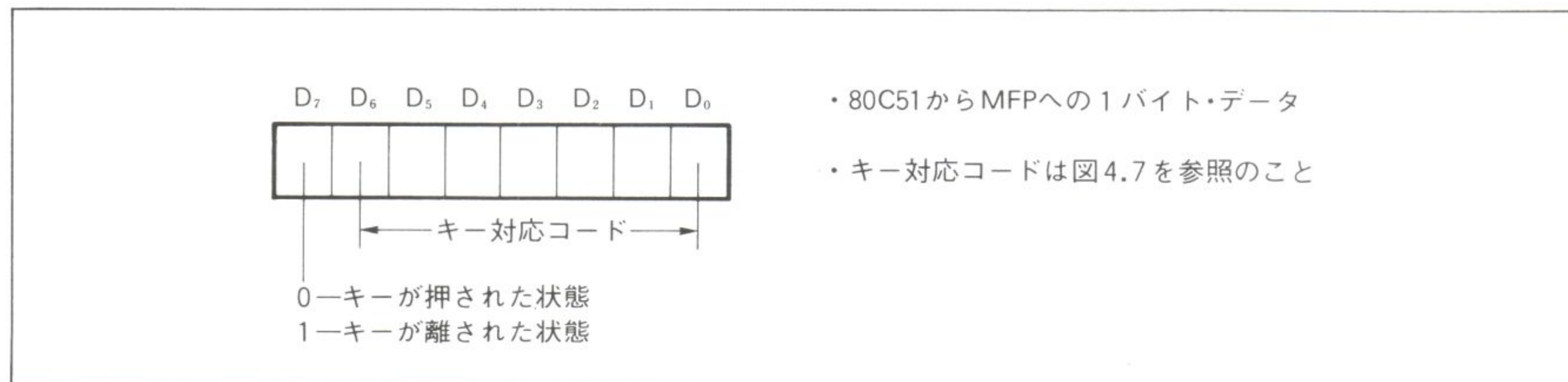
●図4．2　キースキャン・データ・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

D_6〜D_0：キー対応コード

D_7：
0—キーが押された状態
1—キーが離された状態

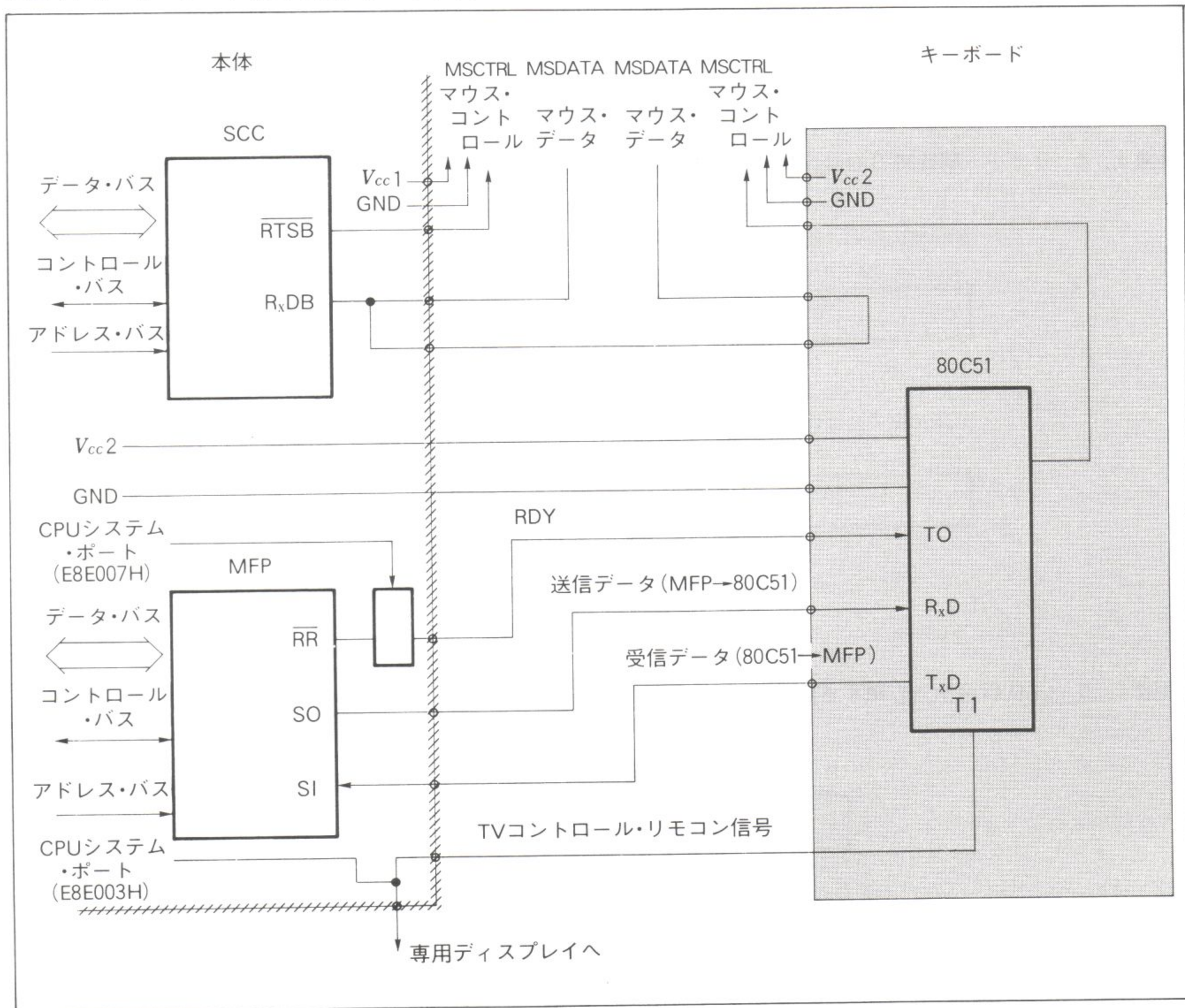
・80C51からMFPへの1バイト・データ

・キー対応コードは図4.7を参照のこと

本章では，前章まで取り上げた以外のデバイスについて説明します．

ここでは，FM音源のように特殊なデバイスについてはその原理にも触れるなど，ダイジェストとはいえ，かなり細かな点にも留意して構成したつもりです．

MFPなどの汎用デバイスについては，紙面の都合もあるので，それぞれのマニュアルを読めばわかる内容については割愛せざるを得ませんでした．ただし，X68000における使われ方という観点で，ユニークな事項については極力掲載することとしました．

●図4．1　キーボードなどの周辺ブロック図

本体
キーボード
MSCTRL マウス・コントロール
MSDATA マウス・データ
MSDATA マウス・データ
MSCTRL マウス・コントロール
SCC
データ・バス
コントロール・バス
アドレス・バス
$\overline{RTSB}$
R_XDB
$V_{CC}1$
GND
$V_{CC}2$
GND
80C51
$V_{CC}2$
GND
CPUシステム・ポート (E8E007H)
RDY
TO
MFP
送信データ(MFP→80C51)
R_XD
データ・バス
$\overline{RR}$
受信データ(80C51→MFP)
T_XD
T1
コントロール・バス
SO
アドレス・バス
SI
TVコントロール・リモコン信号
CPUシステム・ポート (E8E003H)
専用ディスプレイへ

キーボードに対するコマンドのうち最も重要と思われるものは，図4.3の**キー・データ送出禁止コマンド**です．このコマンドはD_0ビットにより，0で禁止，1で許可を与えるようになっています．0にした後は，当然ながらキーボードを押しても入力することはできません．したがって，このコマンドを使用するときは，自動的に復帰できるようなプログラムを作っておかなければなりません．言い換えると，何らかの都合で一時的にキー入力を止めるときにのみ禁止するのが正しい使い方です．

リピート機能は，キーを押して一定時間経ってからさらに押し続けると一定間隔でキーデータの読み取りを行なうものですが，この制御も80C51がやっているので，図4.4のコマンドで時間関係を設定できます．

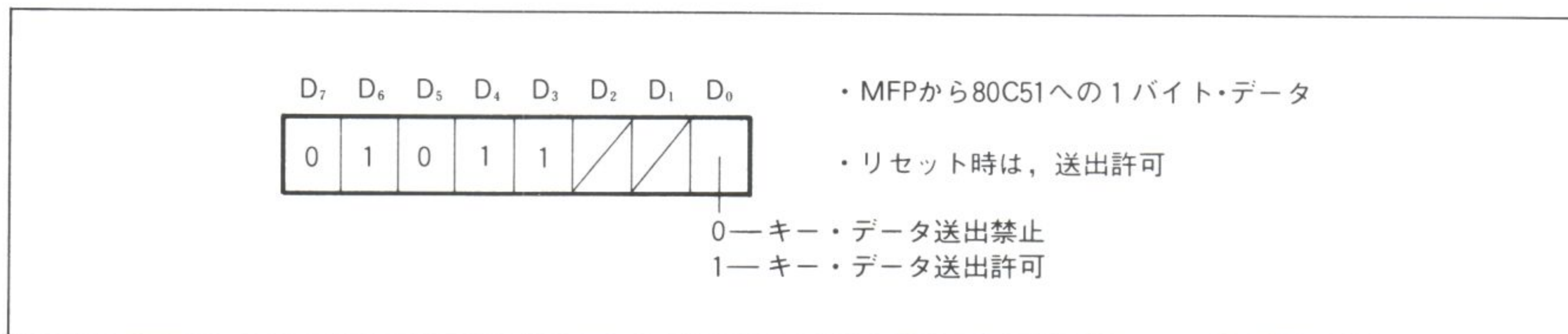
●表4．2　リセット時に同時に押せるキーとLEDの設定

リセット時に押すキー	LEDの明るさ
な　し	最も明るい
XF 3	やや明るい
XF 4	やや暗い
XF 5	暗い

●図4．3　キー・データ送出禁止コマンド・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	1	/	/	

0―キー・データ送出禁止
1―キー・データ送出許可

・MFPから80C51への1バイト・データ
・リセット時は，送出許可

●図4．4　リピート開始時のDelay時間と以降のリピート間隔時間の設定コマンド・フォーマット

（開始時のDelay時間の設定）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	1	0				

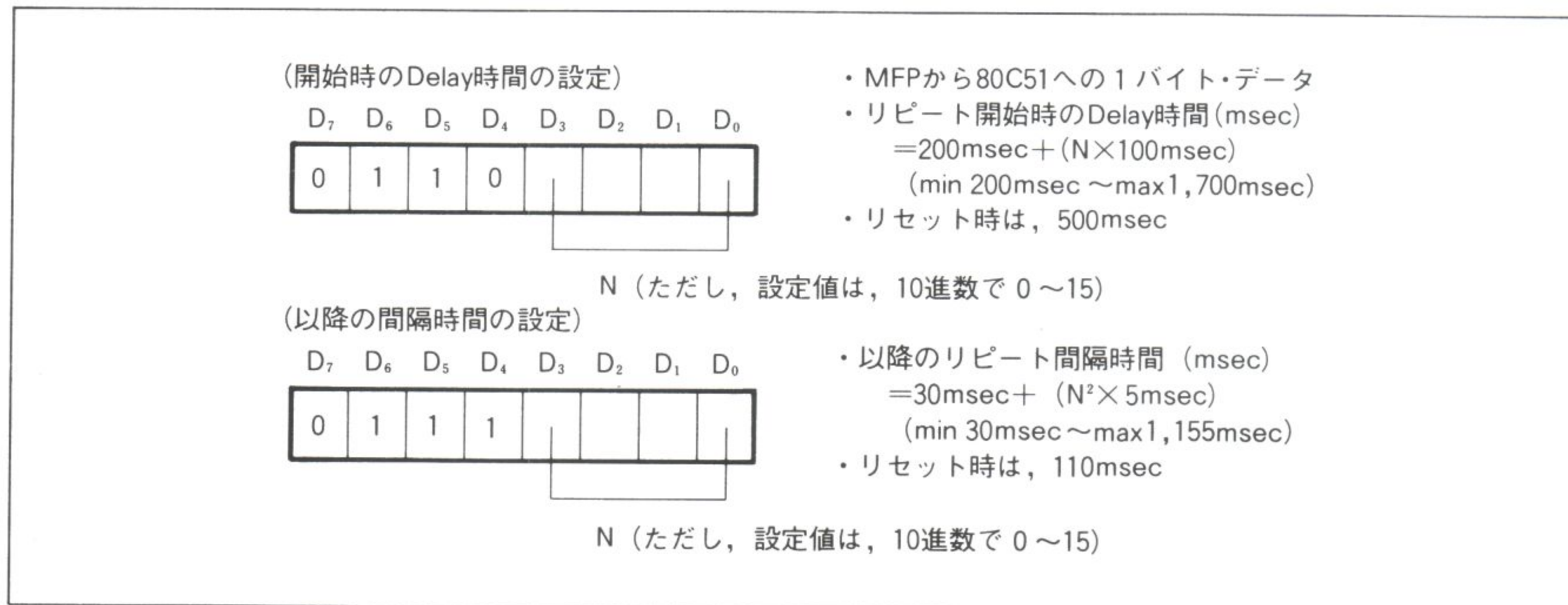
N（ただし，設定値は，10進数で0～15）

・MFPから80C51への1バイト・データ
・リピート開始時のDelay時間（msec）
　＝200msec＋(N×100msec)
　（min 200msec ～max1,700msec）
・リセット時は，500msec

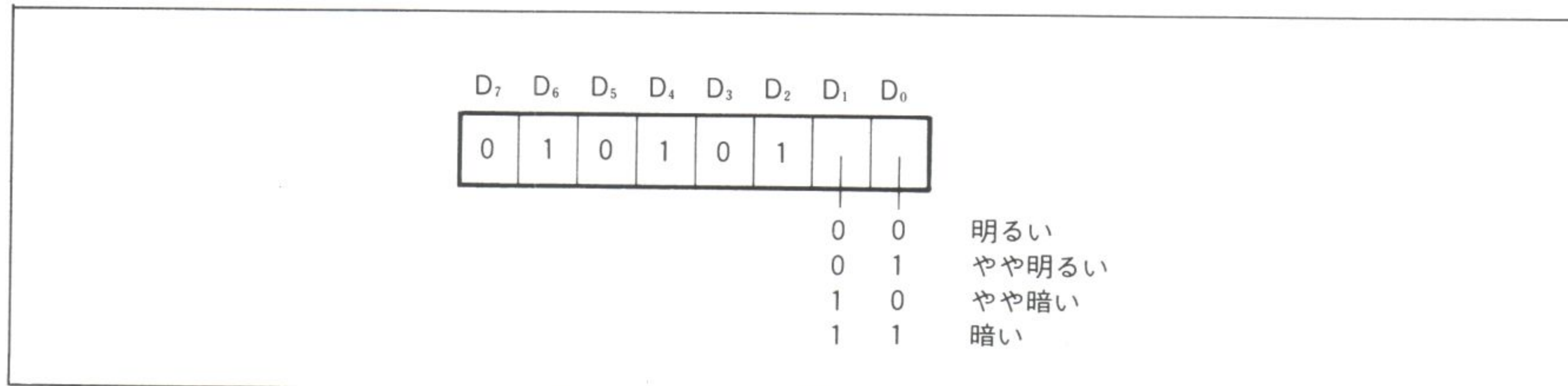
（以降の間隔時間の設定）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	1	1				

N（ただし，設定値は，10進数で0～15）

・以降のリピート間隔時間（msec）
　＝30msec＋（N^2×5msec）
　（min 30msec～max1,155msec）
・リセット時は，110msec

●図4．5　LED付きキーの明るさ調整フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	0	1		

D_1	D_0	
0	0	明るい
0	1	やや明るい
1	0	やや暗い
1	1	暗い

キーボードには「かな」などのLEDが付いています.

LEDの明るさは，リセット時とコマンドにより4段階に調節ができるようになっています．リセット時は表4.2，コマンドによる場合は図4.5のような設定により，任意の明るさにできます.

個々のLEDの点灯，消灯は，図4.6のコマンドで個別に行なえます．このコマンドは全LEDを同時に制御するもので，以前の状態を残す機能はありません．したがって，特定のLEDのみON，OFFしたいときは，メモリの現在のLEDの状況をストアしておき，そのビット構成を操作した上で図のコマンドを送るようにしなければなりません.

なお，キーデータの読み取り時には，図4.7のようなコード体系となっています．したがって，実際に意図した値への変換は，プログラムによって行なわなければなりません.

●図4．6　LED点灯制御コード・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1							

- D_0：0－点灯（かな）／1－消灯（かな）
- D_1：0－点灯（ローマ字）／1－消灯（ローマ字）
- D_2：0－点灯（コード入力）／1－消灯（コード入力）
- D_3：0－点灯（CAPS）／1－消灯（CAPS）
- D_4：0－点灯（INS）／1－消灯（INS）
- D_5：0－消灯（ひらがな）／1－消灯（ひらがな）
- D_6：0－点灯（全角）／1－消灯（全角）

・MFPから80C51への1バイト・データ

・リセット時は，すべて消灯になっており，80C51からLED点灯要求コードFFHが，MFPへ送られる.

●図4．7　各キーの読み取り段階でのコード

61 62　63 64 65 66 67　68 69 6A 6B 6C　5A 5B 5C　5D 52 53 54

01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F　36 5E 37　3F 40 41 42

10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D　38 39 3A　43 44 45 46

71 1E 1F 20 21 22 23 24 25 26 27 28 29　3B 3C 3D 3E　47 48 49 4A

70 2A 2B 2C 2D 2E 2F 30 31 32 33 34 70　4B 4C 4D 4E

5F 55 56 35 57 58 59 60　72 73　4F 50 51

キー・コードは16進数

4-2 リモートTVコントロール

X68000は，キーボード内の80C51に**リモートTVコントロール機能**をもたせてあります．

この機能はキーボードから直接働かせることができるほか，本体内MFPから80C51に制御コード(図4.8)を送って間接的に作動させることもできます．これらの関連は図4.9のとおりで，本体内のORゲートの**コントロール・イネーブル**が0になっているときのみ有効です．なぜなら，この信号が“1”のときは，ORゲートの出力信号は常に“1”となり，TVに信号を送ることができないからです．

TVに送られるシリアル信号は図4.10のような形式で，同じデータをそのままの状態(表信号)と，反転し

●図4.8　TVコントロール・コード・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0						
		←TVコントロール・コード→					

・MFPから80C51へのバイト・データ

・SHIFTキーあるいは，OPT.2＋対応キーで，TVコントロールが行なわれる．

TVコントロール・コード						SHIFTキーあるいは，OPT.2キー＋対応キー	命　　令	機　　能
D_5	D_4	D_3	D_2	D_1	D_0			
*	0	0	0	0	0		――――	
*	0	0	0	0	1	↑	Vol. up	ボリューム・アップ（リピート可）
*	0	0	0	1	0	↓	Vol. down	ボリューム・ダウン（リピート可）
*	0	0	0	1	1	,	Vol. normal	ボリューム・ノーマル
*	0	0	1	0	0	CLR	Call	チャンネル・コール
*	0	0	1	0	1		CS down	テレビ画面（初期化，リセット）
*	0	0	1	1	0	0	Mute	音声ミュート
*	0	0	1	1	1		CH16	
*	0	1	0	0	0	・	BR up	テレビ／コンピュータ画面切り換えのトグル動作
*	0	1	0	0	1	＝	BR down	テレビ／外部入力または，コンピュータ・ノーマル/コンピュータ・オーバースキャン(31.5kHz)切り換えのトグル動作
*	0	1	0	1	0		BR 1/2	コントラスト・ノーマル
*	0	1	0	1	1	→	CH up	チャンネル・アップ（リピート可）
*	0	1	1	0	0	←	CH down	チャンネル・ダウン（リピート可）
*	0	1	1	0	1			
*	0	1	1	1	0		Power ON/OFF	電源ON/OFF
*	0	1	1	1	1	＋	CS 1/2	スーパー1(スーパーインポーズ+コントラスト・ダウン)/スーパー1解除のトグル動作
*	1	0	0	0	0	テンキー1	CH 1	チャンネル1
*	1	0	0	0	1	2	CH 2	チャンネル2
*	1	0	0	1	0	3	CH 3	チャンネル3
*	1	0	0	1	1	4	CH 4	チャンネル4
*	1	0	1	0	0	5	CH 5	チャンネル5
*	1	0	1	0	1	6	CH 6	チャンネル6
*	1	0	1	1	0	7	CH 7	チャンネル7
*	1	0	1	1	1	8	CH 8	チャンネル8
*	1	1	0	0	0	9	CH 9	チャンネル9
*	1	1	0	0	1	／	CH10	チャンネル10
*	1	1	0	1	0	*	CH11	チャンネル11
*	1	1	0	1	1	―	CH12	チャンネル12
*	1	1	1	0	0	(＝)	CH13	テレビ画面（初期化，リセット）
*	1	1	1	0	1	(・)	CH14	コンピュータ画面
*	1	1	1	1	0	(＋)	CH15	スーパー1（スーパーインポーズ+コントラスト・ダウン）
*	1	1	1	1	1			スーパー2（スーパー1＋コントラスト・ノーマル）

注）＊は無効、また、対応キーが何も書かれていない場合は本体からのコード制御のみ可能．
なお，(　)対応キーは，X1コンパチ・モードで有効．

た状態(裏信号)で交互に転送します．このため，80C51が信号を転送していないとき，コントロール・イネーブルをプログラムでON/OFFして図のような信号を作り出し，別途リモートTVコントロールが行なえます．キーボードが接続されていないときは80C51が使えないので，この方法によってカバーするしかありません．

本体からの間接的なTVコントロールを有効にするには，図4.11のコマンドで設定します．また，キー

●図4．9　TVコントロール系統図

本体
E8E003H
(D_3)
MFP
キーボード
80C51
リモートTVコントロール信号
専用TV

●図4．10　リモートTVコントロール信号

(TVコントロール・リモコン信号の波形)
表信号　裏信号
48msec　48msec

(リモコン信号の“0”，“1”の波形)
250μsec　250μsec
1 msec “0”　2 msec “1”
t

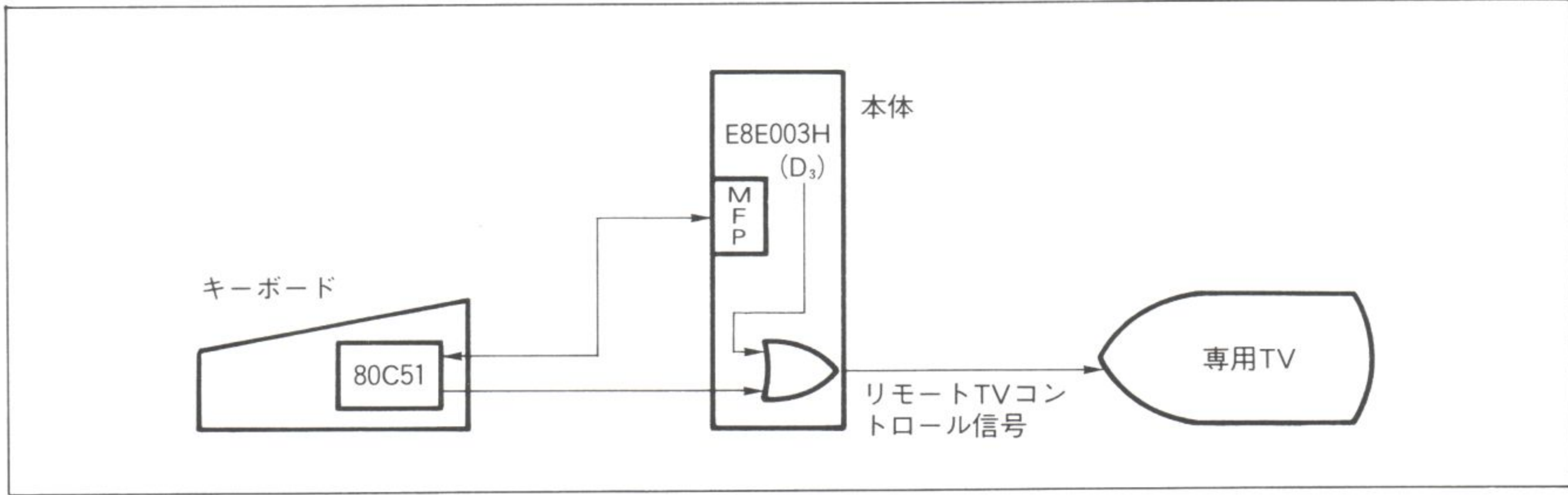
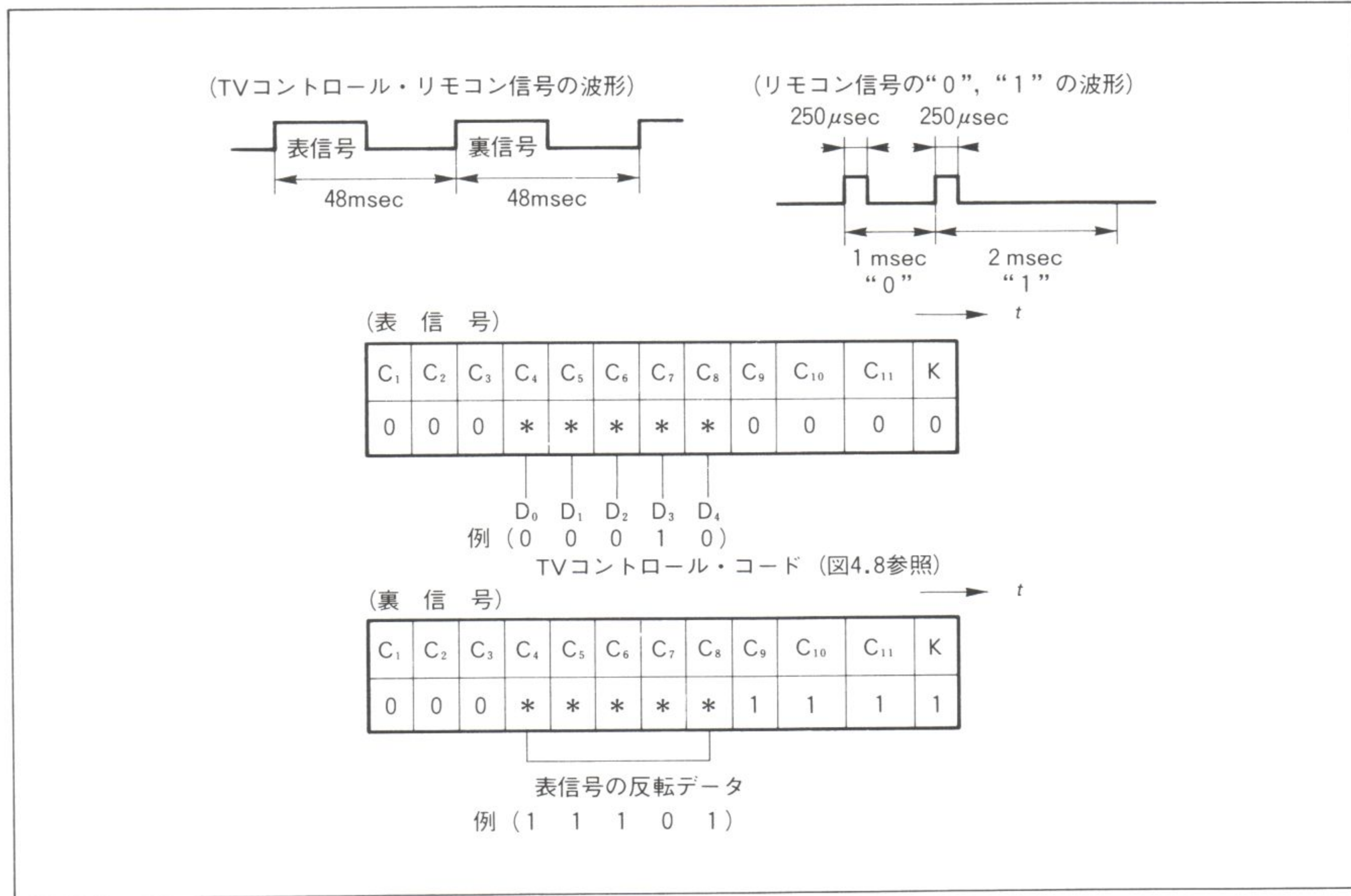
(表　信　号)

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	K
0	0	0	*	*	*	*	*	0	0	0	0

D_0　D_1　D_2　D_3　D_4
例（0　0　0　1　0）
TVコントロール・コード（図4.8参照）

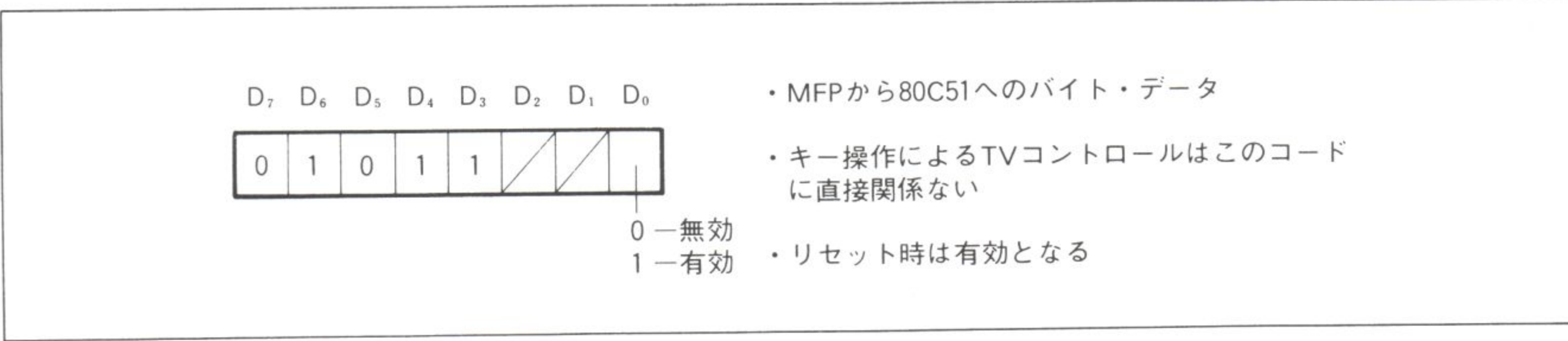
(裏　信　号)

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8	C_9	C_{10}	C_{11}	K
0	0	0	*	*	*	*	*	1	1	1	1

表信号の反転データ
例（1　1　1　0　1）

●図4．11　本体からのTVコントロール・コードの制御コマンド・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	1	/	/	

D_0：0—無効　1—有効

・MFPから80C51へのバイト・データ
・キー操作によるTVコントロールはこのコードに直接関係ない
・リセット時は有効となる

ボード側のTVコントロールについても同様に，図4.12のコマンドで設定できます．なお，X1との互換性をもたせるために，図4.13のコマンドも用意されています．

●図4．12　OPT.キーとテンキーによるTVコントロール制御コード・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	1	1	／	

・MFPから80C51への1バイト・データ
・リセット時は有効

0 ── 有効（TVコントロール・キーとして使用）
1 ── 無効

●図4．13　特定のTVコントロール・キーにおけるX1コンパチ化制御コマンド・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	1	0	／	／	

・MFPから80C51への1バイト・データ
・リセット時はX1コンパチ解除
・X 1コンパチのキー操作は表4.3参照

0 ― X1コンパチ
1 ― X1コンパチ解除

●表4．3　X1コンパチのキー操作

操　作　キ　ー	＃1000	X1コンパチ・モード
SHIFT ＋ ＋	スーパーインポーズ/スーパーインポーズ解除のトグル・キー	スーパーインポーズ
SHIFT ＋ ＝	TV／外部入力切り換えのトグル・キー	TV
SHIFT ＋ ・	TV／コンピュータ画面切り換えのトグル・キー	コンピュータ

4-3 マウスとのインターフェース

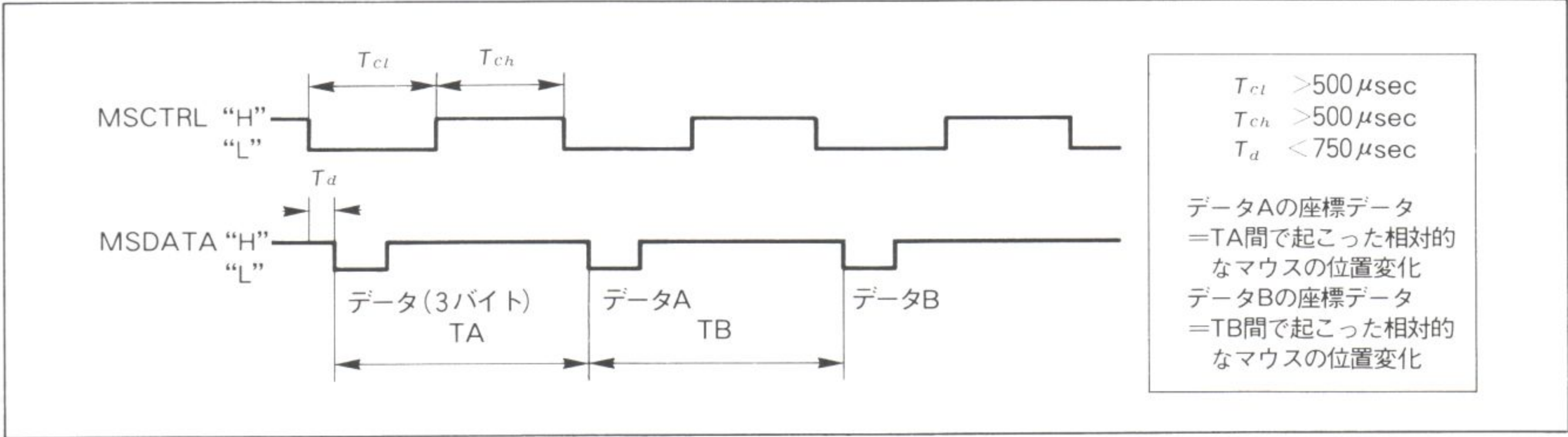
本体からの転送要求に対し，前後方向(Y)，左右方向(X)の移動量およびスイッチの情報を転送するのがマウスの動作です．X方向，Y方向，スイッチの定義を図4.14に示します．

マウスに対するデータの転送要求信号と応答信号との関係は，図4.15のとおりで，移動量の転送が始まるとカウンタはクリアされ，次回のための移動量のカウントを始めます．転送時の手順は調歩式で図4.16のような規格によっています．このような制御をするため，本体側ではSCCに接続されています．

転送データは3バイト(図4.17)で，1バイト目はスイッチ情報とともにエラー情報を含んでいます．エラーとは，カウント・アップまたはカウント・ダウン時の**オーバーフロー**(正方向の行き過ぎ)，**アンダーフロー**(負方向の行き過ぎ)のことです．このような状態になると正しくトレースできなくなるので，プログラムではかなり頻繁にデータの読み取りを行ない，エラーが生じないようにしなければなりません．

マウスは本体以外にキーボードのマウス・コネクタにも接続できますが，このときのMSCTRL(マウス・コントロール)制御は，図4.18のコマンドを使って行ないます．

●図4．14　マウスの方向およびスイッチの定義

●図4．15　マウスに対する転送要求と応答信号

●図4．16　マウス・データ通信手順とデータ転送手順

マウス・データ・フォーマット	通信モード	非同期通信
	ボーレート	4,800ボー
	スタート・ビット	1
	データ長	8
	ストップ・ビット	2

端子番号	信号名
1	V_{CC}
2	MSCTRL
3	MSDATA
4	GND
5	GND

●図4．17　マウス・データ・フォーマット

マウス・データ (MSDATA)	1バイト目	ステータス・データ
	2バイト目	X方向データ　－128 ～ ＋127
	3バイト目	Y方向データ　－128 ～ ＋127

〈ステータス・データ〉

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
				0	0		

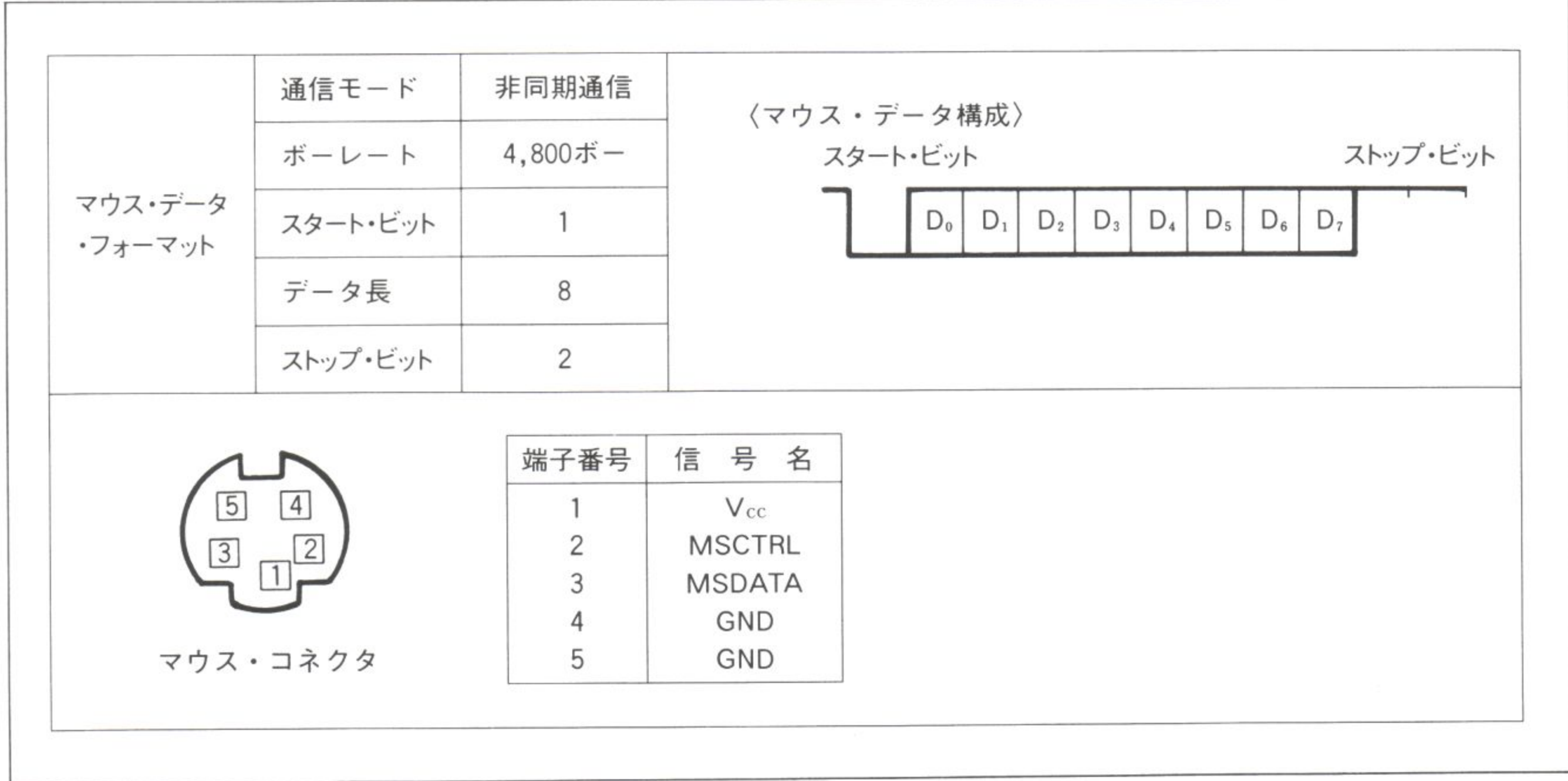
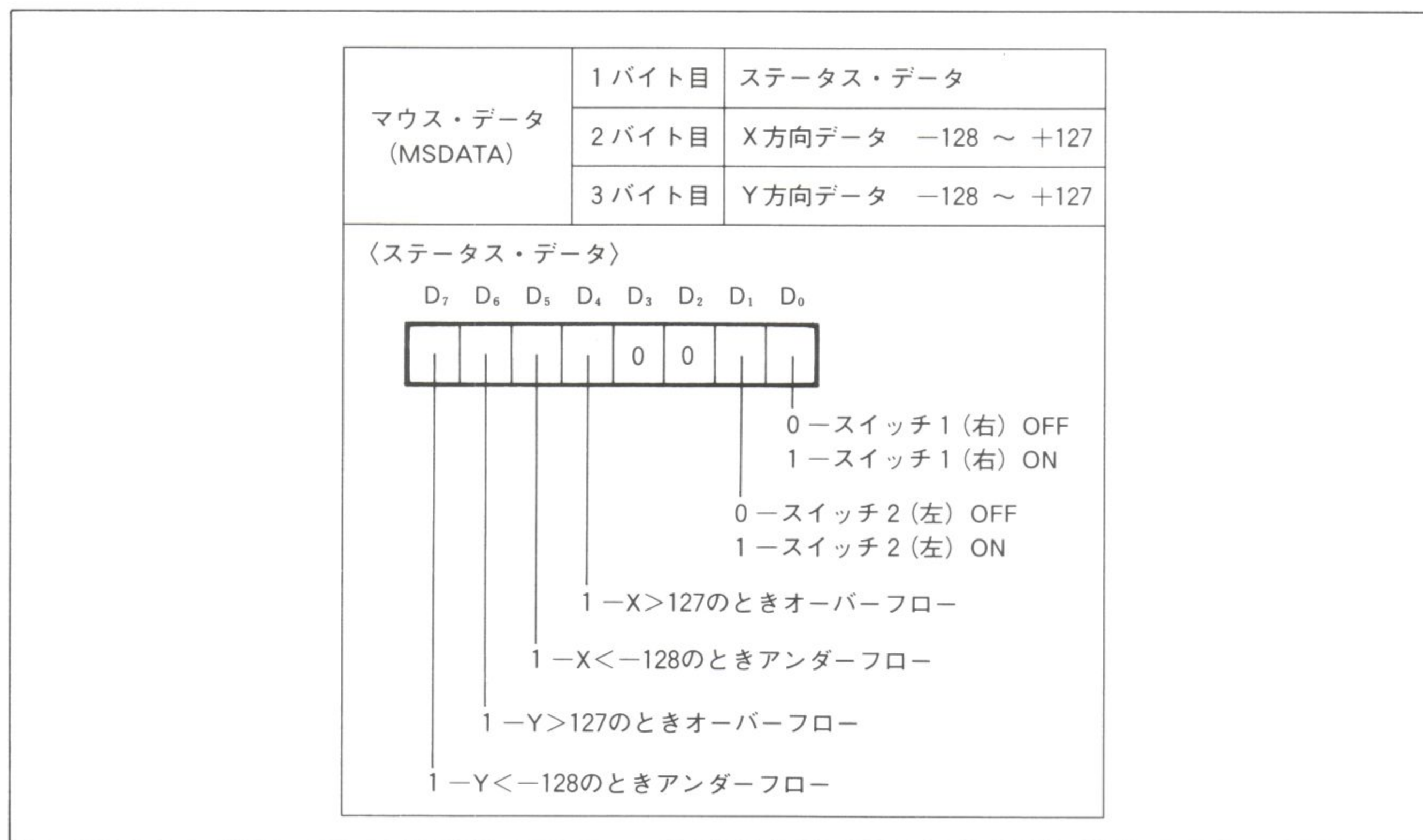
D_0：0－スイッチ1（右）OFF　1－スイッチ1（右）ON

D_1：0－スイッチ2（左）OFF　1－スイッチ2（左）ON

D_4：1－X＞127のときオーバーフロー

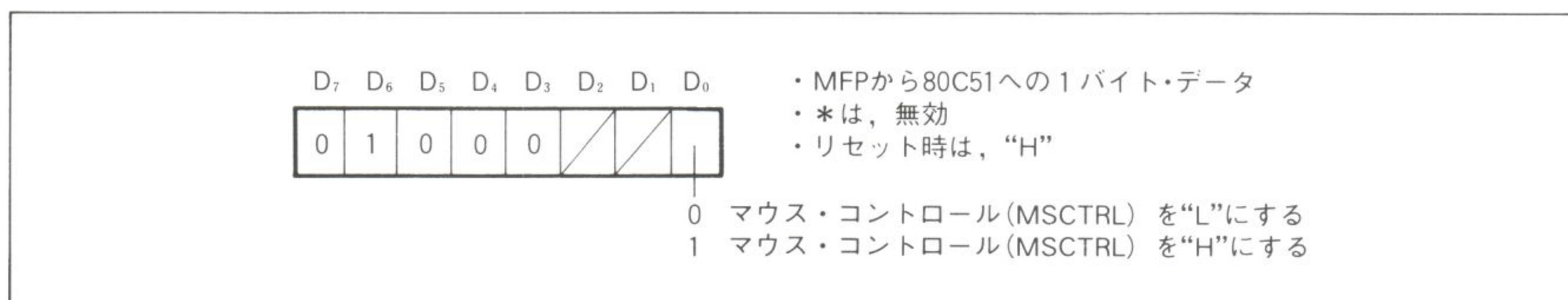
D_5：1－X＜－128のときアンダーフロー

D_6：1－Y＞127のときオーバーフロー

D_7：1－Y＜－128のときアンダーフロー

●図4．18　マウス・コントロール制御コマンド・フォーマット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	1	0	0	0	/	/	

・MFPから80C51への1バイト・データ
・＊は，無効
・リセット時は，“H”

D_0：0　マウス・コントロール(MSCTRL)　を“L”にする
1　マウス・コントロール(MSCTRL)　を“H”にする

4-4　サウンド機能の概要

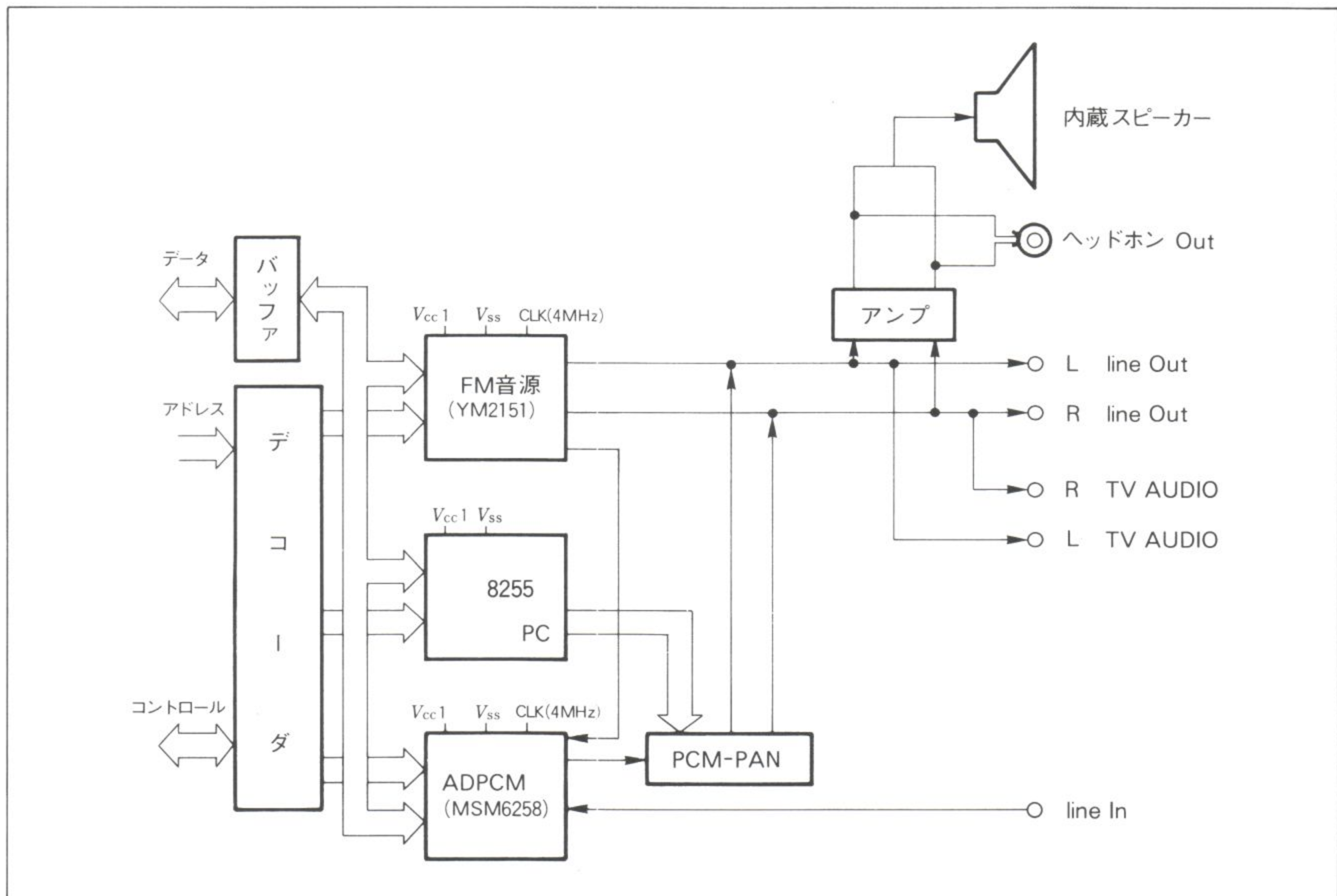
X68000のサウンド系のブロック図を図4.19に示します．

これらを機能面で大別すると，**FM音源**，**音声合成**に分けられますが，FM音源IC(YM2151)の一部(データ・ポート)は，音声合成用にも使われています．FM音源はステレオ用に設計されており，8チャンネルの楽音を任意に左右(L，R)に接続できます．

PCM音源は，入力端子から与えられた音声信号を，**ADPCM**方式で圧縮し，PCM化してメモリに記録します．再生する場合は逆変換を行ない，**PCM-PAN**経由で音声ラインに出力します．

ただし，**ヘッドホン**ではステレオでエンジョイできるものの，**内蔵スピーカー**は1個しかなく，左右の混合された音となります．したがって，スピーカーでステレオ音を楽しむには，**ラインアウト**端子から別途ステレオ・アンプを経由して外付けスピーカーを駆動するしかありません．

もっとも，内蔵アンプはパワーが少なく，それだけでは大音量を発生することができないので，いわばオマケのようなものと考えた方がよいでしょう．それでも，何かのエラーのとき警告音を出す際には充分役に立ちます．

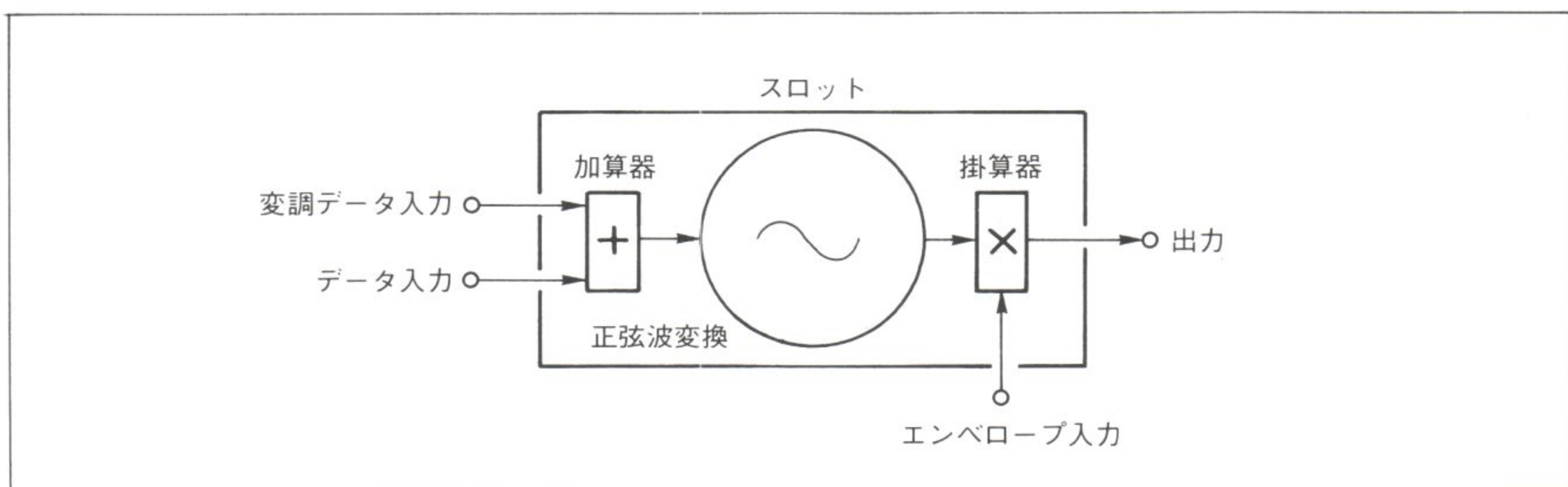
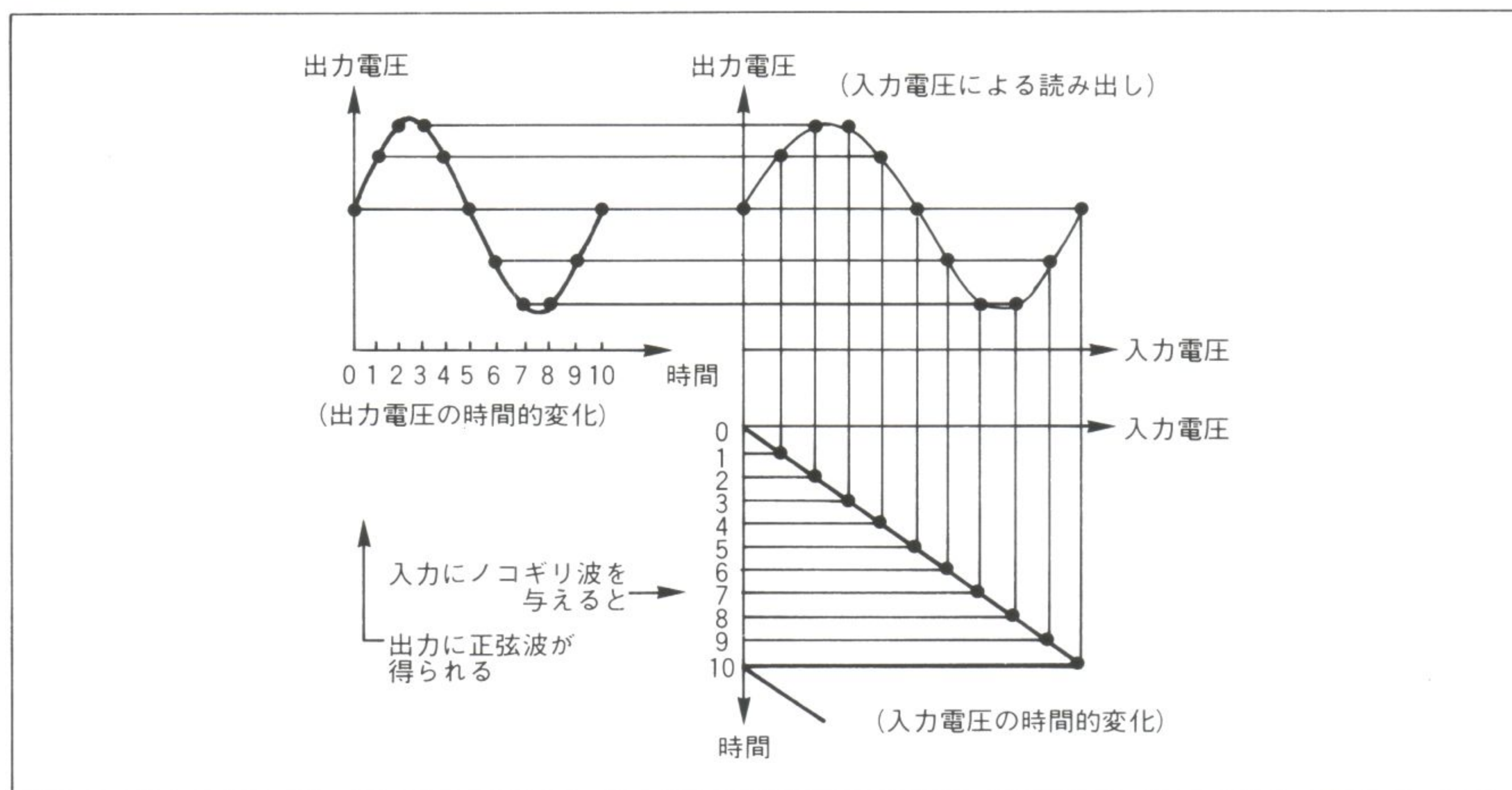
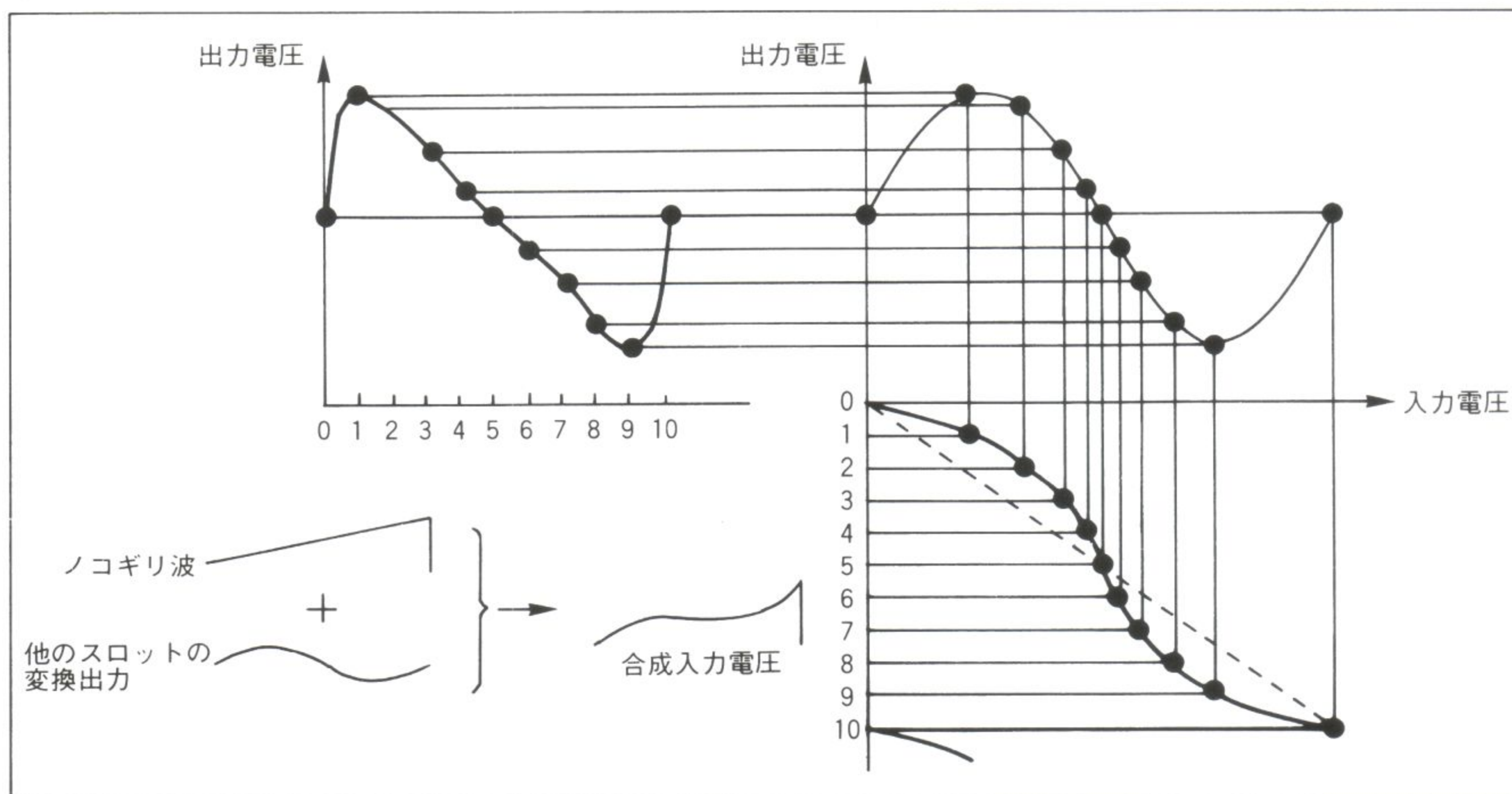
●図4．19　サウンド系ブロック図

4-5 FM音源の動作原理

FM音源の最も中心となる部分は，**スロット**です．スロットは，図4.20のような構造をしていて，入力データとして**ノコギリ波**を与えると，図4.21の対応付けによって，**正弦波**を出力するしかけになっています．いわば，入力データによってROMを読み出すような概念です．

この仕組を利用して，ノコギリ波と他の波形とを混合して入力すると，出力波形は図4.22のように複雑な形に変わります．この原理で音作りをします．

波形の混ぜ方を**アルゴリズム**といい，図4.23のようなパターンが用意されています．各パターンにおいて，スロット1はその出力を入力側に**フィード・バック**できます．フィード・バックは，金管楽器的な音作りに効果があります．

スロットの**エンベロープ**入力は，出力の振幅調整に用いられます．これによって，図4.24のように楽器

●図4．20　スロットの構成

●図4．21　スロットにノコギリ波を入力したときの出力の状況

●図4．22　合成波形をスロットに入力したときの動作例

音の立ち上がりに該当する**アタック**，直後の**ディケイ**，余韻にあたる**サスティン**，止めの**リリース**の各段階ごとに時間間隔を設定できます．また，サスティンでの落ち込みの程度は，**サスティン・レベル**として設定することになっています．

たとえば，打楽器などはアタックが鋭く，またサスティンの落ち込みも大きくなりますが，バイオリンなどではアタックが甘く，ほとんど落ち込まないでサスティンに入ります．いわば音の強弱の変化を利用して，楽器らしさを演出するのです．

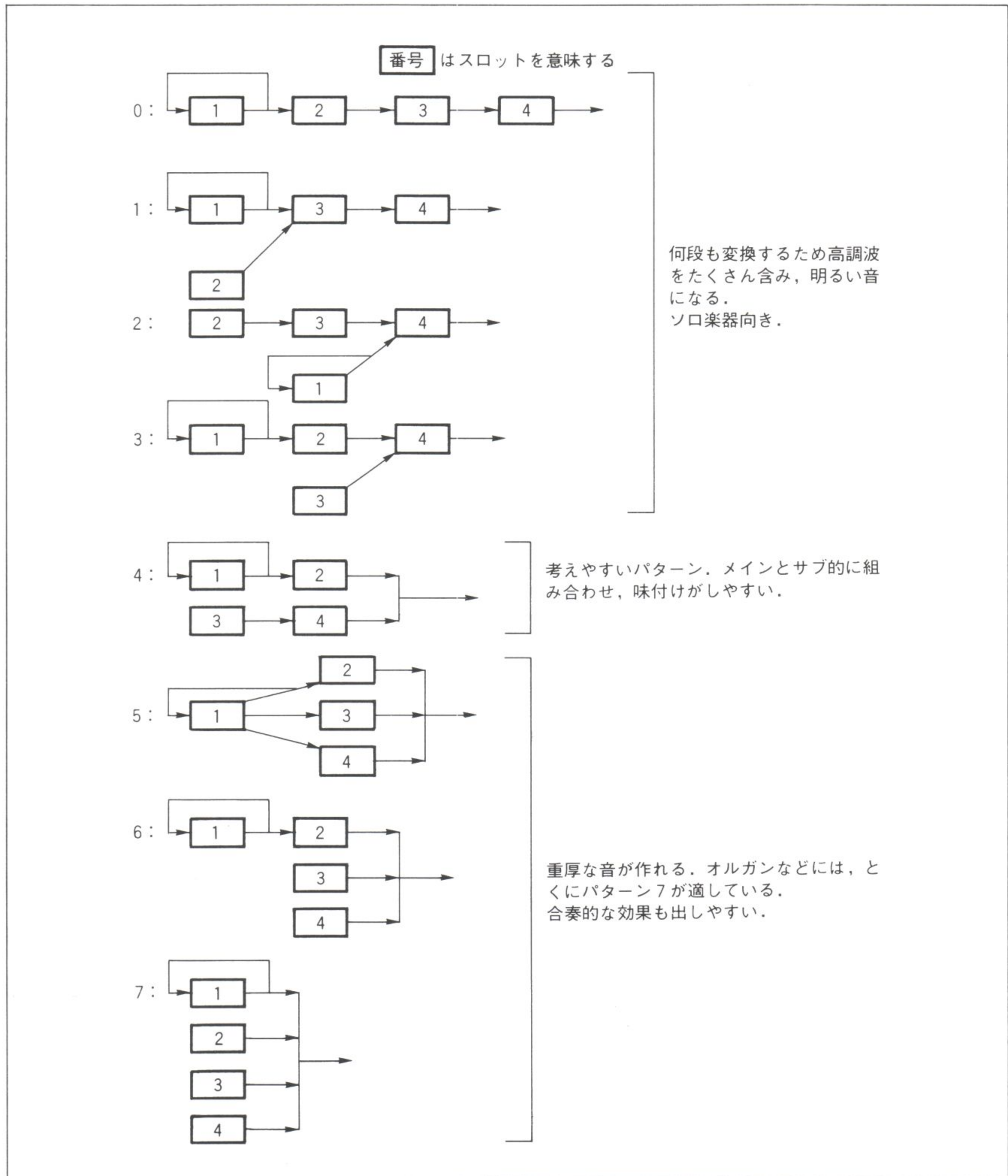
FM音源では，周波数が非常に正確で安定しています．ところが実際の楽器では，演奏中に微妙にズレたりして，いわばこれが演奏者の持ち味になっている面もあります．同じ楽譜で演奏しても，こういったデリケートなところで人間と機械との差が出てしまうのです．そこで，**デチューン**によって，故意に調子をはずすことができるようになっています．また，音のゆらぎは，**LFO**(超低周波発振器)で作り出すことができ，同様にLFOを出力レベルの制御に用いてビブラート効果を出すことも可能です．

●図4．23　アルゴリズムのパターン

番号 はスロットを意味する

何段も変換するため高調波をたくさん含み，明るい音になる．ソロ楽器向き．

考えやすいパターン．メインとサブ的に組み合わせ，味付けがしやすい．

重厚な音が作れる．オルガンなどには，とくにパターン7が適している．合奏的な効果も出しやすい．

音源用IC(YM2151＝OPMなど)のブロック図は図4.25のとおりで，4個のスロットの組み合わせを，計8組もっています．言い換えると，8音まで出せることになります．アルゴリズム・パターン7を使えば，物理的に正弦波で32音まで出すことも不可能ではありません．

各スロットの組の出力はそれぞれ任意にL(左)，R(右)，または両側に振り分けられ，ステレオ演奏ができるのもOPMの特徴です．

●図4．24　FM音源のエンベロープ・コントロール

出力レベル
トータル・レベル
サスティン・レベル
時間
アタック（TA）
ディケイ（TD 1）
サスティン（TD 2）
リリース（TR）
キー ON
キー OFF

●図4．25　FM音源（YM-2151）ブロック図

各Slotブロック図
PG　Sin Table　EG

Ch. 1　M 1：Slot. 1　M 1：Slot. 9　C 1：Slot. 17　C 2：Slot. 25　OPerator
Ch. 2　M 1：Slot. 2　M 2：Slot. 10　C 1：Slot. 18　C 2：Slot. 26　OP
Ch. 3　M 1：Slot. 3　M 2：Slot. 11　C 1：Slot. 19　C 2：Slot. 27　OP
Ch. 4　M 1：Slot. 4　M 2：Slot. 12　C 1：Slot. 20　C 2：Slot. 28　OP
Ch. 5　M 1：Slot. 5　M 2：Slot. 13　C 1：Slot. 21　C 2：Slot. 29　OP
Ch. 6　M 1：Slot. 6　M 2：Slot. 14　C 1：Slot. 22　C 2：Slot. 30　OP
Ch. 7　M 1：Slot. 7　M 2：Slot. 15　C 1：Slot. 23　C 2：Slot. 31　OP
Ch. 8　M 1：Slot. 8　M 2：Slot. 16　C 1：Slot. 24　C 2：Slot. 32　OP

Noise Slotと共用

OPM YM2151　LFO　ACC
YM3012　DAC　L　R

*各チャンネル動作は時分割制御

M 1 ：Modulator 1
M 2 ：Modulator 2
C 1 ：Carrier 1
C 2 ：Carrier 2
OP ：FM Operator
LFO ：Low Frequency Oscillator
ACC ：Accumulator
PG ：Phase Generator
EG ：Envelope Generator

4-6 FM音源のレジスタ構成と内部/外部アドレス・マップ

FM音源の内部には各種レジスタがあって，CPUからデータを送り込んで書き込むことができます．この場合，どのレジスタを選択するかの指定を行なうため**レジスタ・アドレス**が使われ，最初に内部アドレ

●表4．4　FM音源のレジスタ・アドレス

レジスタ・アドレス	Read	Write
E90001H	——	内部レジスタ・アドレス
E90003H	フラグ・レジスタ	レジスタ・データ

●表4．5　FM音源構成レジスタの機能概要

項目	レジスタ名		機能
レジスタ(WRITE MODE)	CT		出力端子CT1，CT2に対応し，外部制御を行なう．
	KON		チャンネル・ナンバーとスロット・ナンバーにキーON/OFFする．
	PG (PHASE GENERATOR)	KC	OCT(8オクターブ)とNOTEの設定を行なう．
		KF	1.6セント刻みの音程差で位相情報を与える．
		MUL	KC，KFの位相情報の倍率を定める．
		DT1	わずかに周波数のずれた位相情報(KCによるスケーリング)を与える．
		DT2	非常に大きなずれの位相情報を与える．
		PMS	ビブラート，ゆらぎのある音などの効果音に有効な位相情報を与える．
	EG (ENVELOPE GENERATOR)	AR	アタック時間(TA)の設定．
		D1R	ファスト・ディケイ時間(TD1)の設定．KCによりスケーリング可．
		D2R	セカンド・ディケイ時間(TD2)の設定．KCによりスケーリング可．
		RR	リリース時間(TR)の設定．KCによりスケーリング可．
		KS	AR，D1R，D2R，RRのスケーリング・レベルをコントロールする．
		D1L	EGがファースト・ディケイからセカンド・ディケイに移行するレベルの設定．
		TL	音色(変調度)および音量を制御するためのトータル・レベル．
		AMS	振幅変調の設定．
	OP (FM OPERATOR)	CON	FM-OPの回路構成を設定する．(8種類)
		FL	位相情報にフィード・バックするレベルのコントロールを行なう．
	NOISE (NOISE GENERATOR)	NE	ノイズの設定．
		NFRQ	ノイズの周期設定．
	LFO (LOW FREQUENCY OSCILLATOR)	LFRQ	発振周波数の設定．
		W	周波数(PM)，振幅(AM)変調用の波形．
		PMD	周波数変調用信号の出力レベルの制御．
		AMD	振幅変調用信号の出力レベルの制御．
	ACC	LR	OPから信号をL系列，R系列に，または両系列同時にアキュムレートする．
	TIMER	CLKA1, CLKA2	タイマAにオーバーフローを発生させる．
		CLKB	タイマBにオーバーフローを発生させる．
		LOAD	タイマA，Bの始動，停止の制御．
		F RESET	オーバーフロー発生を示すフラグ・レジスタの内容をリセットする．
		IRQEN	タイマが発生するオーバーフローをレジストし，割り込みを可能にする．
		CSM	タイマAのオーバーフロー発生時に音源すべてのスロットをキーONする．(被合正弦波音声合成が可.)
フラグ・レジスタ(READ MODE)	B		レジスタ・データ書き込み中であることを示すフラグ．
	IST		タイマA，Bのフラグの状態を示す．

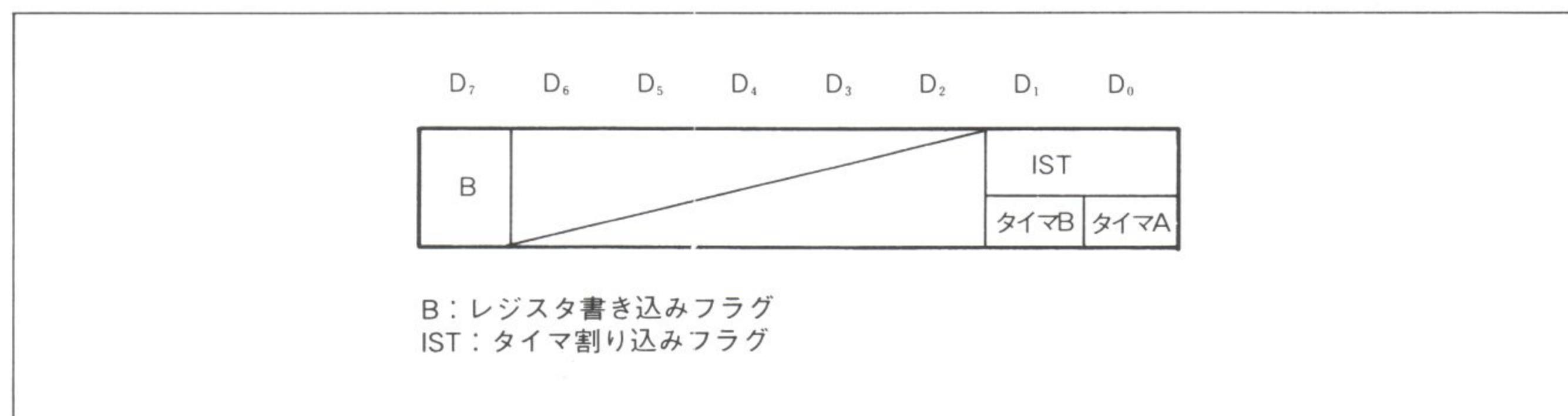
スをセットして，次にデータ値を転送する手順を踏みます．この方法によれば，直接レジスタに書き込むシステムに比べ，システム・バス上の占有アドレス幅を少なくできます．

●図4．26　FM音源の内部アドレスの割り付け

内部アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備　考
01H	TEST								D_1はLFO RESET（普通は0とする）
08H	×	KON							D_3-D_6はスロットNo.，D_0-D_2はチャンネルNo.
0FH	NE	×	×	NFRQ					
10H	CLKA1								
11H	×	×	×	×	×	×	CLKA2		
12H	CLKB								
14H	CSM	×	F RESET		IRQEN		LOAD		
18H	LFRQ								
19H	PMD/AMD								
1BH	CT		×	×	×	×	W		
20H	LR		FL			CON			チャンネル1
〜	〜		〜			〜			〜
27H	LR		FL			CON			チャンネル8
28H	×	KC							チャンネル1 （D_4-D_6はオクターブ，D_0-D_3はNOTE）
〜	〜	〜							
2FH	×	KC							チャンネル8 （D_4-D_6はオクターブ，D_0-D_3はNOTE）
30H	KF						×	×	チャンネル1
〜	〜						〜		〜　　チャンネル単位
37H	KF						×	×	チャンネル8
38H	×	PMS			×	×	AMS		チャンネル1
〜	〜	〜			〜		〜		〜
3FH	×	PMS			×	×	AMS		チャンネル8
40H	×	DT1			MUL				スロット1
〜	〜	〜			〜				〜
5FH	×	DT1			MUL				スロット32
60H	×	TL							スロット1
〜	〜								〜
7FH	×	TL							スロット32　　スロット単位
80H	KS		×	AR					スロット1
〜	〜		〜	〜					〜
9FH	KS		×	AR					スロット32
A0H	AMS	×	×	DIR					スロット1（D_7はAMS-EN）
〜	〜	〜		〜					〜
BFH	AMS	×	×	DIR					スロット32（D_7はAMS-EN）
C0H	DT2		×	D2R					スロット1
〜	〜		〜	〜					〜
DFH	DT2		×	D2R					スロット32　　スロット単位
E0H	D1L				RR				スロット1
〜	〜				〜				〜
FFH	D1L				RR				スロット32

以上のレジスタは書き込みのみですが，FM音源では内部の動作状態の一部をCPUから監視できるように，リード・オンリーの**フラグ・レジスタ**をもっています．これらのシステム・バス上のアドレス配置は表4.4のとおりです．

個々のレジスタの機能の概要は表4.5のようなもので，内部アドレスの割り付けは図4.26のとおりになっています．フラグ・レジスタのビット構成は図4.27のとおりです．

●図4．27　フラグ・レジスタのビット構成

4-7 FM音源の個々のレジスタ設定

FM音源のレジスタ設定は大別して，楽音の質，音程，修飾(ビブラートなど)といった要素のほかに，発音タイミングの制御が加わります．

このうち，「音作り」は始めに行ない，あとはそのまま引き継ぐのがほとんどです．この分類に属するパラメータは，**CON**がその代表的なものです．一般にFM音源はレジスタ数が多いため，曲の場面に応じてこと細かく設定するのが大変です．そこで，面倒を省くため，ビブラートなどの修飾も，最初に設定したらあとはかけっ放しというケースが多いようです．

個々の発音に関しては，音程が最も大事なパラメータとなり，このため**KC**，**KF**を中心にレジスタ設定を行なうことになります．音長はキーONからOFFまでで決まりますが，この時間間隔の設定には，**タイマ**を使うのが最も簡単です．したがって，発音前に必要なパラメータをすべてセットしてキーONし，タイマによる割り込みでキーOFFという手順を踏むのが一般的です．

個別のパラメータの具体的なビット構成，設定値の意味などは，図4.28のとおりです．

●図4．28　FM音源のパラメータ設定①

（続く）

●図4．28　FM音源のパラメータ設定②

	D7	D6	D5	D4	D3	D2	D1	D0
28H〜2FH		OCT			NOTE			
		オクターブ			ノート			

	D7	D6	D5	D4	D3	D2	D1	D0
30H〜37H	KF							
	音程微調整							

これらの関係は付図2参照

	D7	D6	D5	D4	D3	D2	D1	D0
38H〜3FH		PMS					AMS	
		位相変調感度（付表3参照）					振幅変調感度（付表4参照）	

	D7	D6	D5	D4	D3	D2	D1	D0
40H〜5FH		DT 1			MUL			
		符号	FD					
		周波数ズレ（小）（付表5参照）			KCFの倍率（付表6参照）			

	D7	D6	D5	D4	D3	D2	D1	D0
60H〜7FH		TL						
		48	24	12	6	3	1.5	0.75
		トータル・レベル(dB)						

	D7	D6	D5	D4	D3	D2	D1	D0
80H〜9FH	KS			AR				
	キー・スケーリング（付表7参照）			アタック・レート（付表8参照）				

	D7	D6	D5	D4	D3	D2	D1	D0
A0H〜BFH	AMS			D1R				
	振幅変調スイッチ（1＝ON，0＝OFF）			ファースト・ディケイ・レート(TD1)（付表9参照）				

	D7	D6	D5	D4	D3	D2	D1	D0
C0H〜DFH	DT2			D2R				
	周波数ズレ(大)（付表10参照）			セカンド・ディケイ・レート(TD2)（付表9参照）				

	D7	D6	D5	D4	D3	D2	D1	D0
E0H〜FFH	D1L				RR			
	ファースト・ディケイ・レベル（付表11参照）				リリース・レート（付表7参照）			

●付表1　LFOの発振周波数

LFREQ (HEX)	周波数 (Hz)	LFREQ (HEX)	周波数 (Hz)	LFREQ (HEX)	周波数 (Hz)	LFREQ (HEX)	周波数 (Hz)
F F	59.1278	B F	3.6955	7 F	0.2310	3 F	0.0144
F E	57.2205	B E	3.5763	7 E	0.2235	3 E	0.0140
F D	55.3131	B D	3.4571	7 D	0.2161	3 D	0.0135
F C	53.4058	B C	3.3379	7 C	0.2086	3 C	0.0130
F B	51.4984	B B	3.2187	7 B	0.2012	3 B	0.0126
F A	49.5911	B A	3.0994	7 A	0.1937	3 A	0.0121
F 9	47.6837	B 9	2.9802	7 9	0.1863	3 9	0.0116
F 8	45.7764	B 8	2.8610	7 8	0.1788	3 8	0.0112
F 7	43.8690	B 7	2.7418	7 7	0.1714	3 7	0.0107
F 6	41.9617	B 6	2.6226	7 6	0.1639	3 6	0.0102
F 5	40.0543	B 5	2.5034	7 5	0.1565	3 5	0.0098
F 4	38.1470	B 4	2.3842	7 4	0.1490	3 4	0.0093
F 3	36.2396	B 3	2.2650	7 3	0.1416	3 3	0.0088
F 2	34.3323	B 2	2.1458	7 2	0.1341	3 2	0.0084
F 1	32.4249	B 1	2.0266	7 1	0.1267	3 1	0.0079
F 0	30.5176	B 0	1.9073	7 0	0.1192	3 0	0.0075
E F	29.5639	A F	1.8477	6 F	0.1155	2 F	0.0072
E E	28.6102	A E	1.7881	6 E	0.1118	2 E	0.0070
E D	27.6566	A D	1.7285	6 D	0.1080	2 D	0.0068
E C	26.7029	A C	1.6689	6 C	0.1043	2 C	0.0065
E B	25.7492	A B	1.6093	6 B	0.1006	2 B	0.0063
E A	24.7955	A A	1.5497	6 A	0.0969	2 A	0.0061
E 9	23.8419	A 9	1.4901	6 9	0.0931	2 9	0.0058
E 8	22.8882	A 8	1.4305	6 8	0.0894	2 8	0.0056
E 7	21.9345	A 7	1.3709	6 7	0.0857	2 7	0.0054
E 6	20.9808	A 6	1.3113	6 6	0.0820	2 6	0.0051
E 5	20.0272	A 5	1.2517	6 5	0.0782	2 5	0.0049
E 4	19.0735	A 4	1.1921	6 4	0.0745	2 4	0.0047
E 3	18.1198	A 3	1.1325	6 3	0.0708	2 3	0.0044
E 2	17.1661	A 2	1.0729	6 2	0.0671	2 2	0.0042
E 1	16.2125	A 1	1.0133	6 1	0.0633	2 1	0.0040
E 0	15.2588	A 0	0.9537	6 0	0.0596	2 0	0.0037
D F	14.7820	9 F	0.9239	5 F	0.0577	1 F	0.0036
D E	14.3051	9 E	0.8941	5 E	0.0559	1 E	0.0035
D D	13.8283	9 D	0.8643	5 D	0.0540	1 D	0.0034
D C	13.3514	9 C	0.8345	5 C	0.0522	1 C	0.0033
D B	12.8746	9 B	0.8047	5 B	0.0503	1 B	0.0031
D A	12.3978	9 A	0.7749	5 A	0.0484	1 A	0.0030
D 9	11.9209	9 9	0.7451	5 9	0.0466	1 9	0.0029
D 8	11.4441	9 8	0.7153	5 8	0.0447	1 8	0.0028
D 7	10.9673	9 7	0.6855	5 7	0.0428	1 7	0.0027
D 6	10.4904	9 6	0.6557	5 6	0.0410	1 6	0.0026
D 5	10.0136	9 5	0.6258	5 5	0.0391	1 5	0.0024
D 4	9.5367	9 4	0.5960	5 4	0.0373	1 4	0.0023
D 3	9.0599	9 3	0.5662	5 3	0.0354	1 3	0.0022
D 2	8.5831	9 2	0.5364	5 2	0.0335	1 2	0.0021
D 1	8.1062	9 1	0.5066	5 1	0.0317	1 1	0.0020
D 0	7.6294	9 0	0.4768	5 0	0.0298	1 0	0.0019
C F	7.3918	8 F	0.4619	4 F	0.0289	0 F	0.0018
C E	7.1526	8 E	0.4470	4 E	0.0279	0 E	0.0017
C D	6.9141	8 D	0.4321	4 D	0.0270	0 D	0.0017
C C	6.6757	8 C	0.4172	4 C	0.0261	0 C	0.0016
C B	6.4373	8 B	0.4023	4 B	0.0251	0 B	0.0016
C A	6.1989	8 A	0.3874	4 A	0.0242	0 A	0.0015
C 9	5.9605	8 9	0.3725	4 9	0.0233	0 9	0.0015
C 8	5.7220	8 8	0.3576	4 8	0.0224	0 8	0.0014
C 7	5.4836	8 7	0.3427	4 7	0.0214	0 7	0.0013
C 6	5.2452	8 6	0.3278	4 6	0.0205	0 6	0.0013
C 5	5.0068	8 5	0.3129	4 5	0.0196	0 5	0.0012
C 4	4.7684	8 4	0.2980	4 4	0.0186	0 4	0.0012
C 3	4.5300	8 3	0.2831	4 3	0.0177	0 3	0.0011
C 2	4.2915	8 2	0.2682	4 2	0.0168	0 2	0.0010
C 1	4.0531	8 1	0.2533	4 1	0.0158	0 1	0.0010
C 0	3.8147	8 0	0.2384	4 0	0.0149	0 0	0.0009

●付表2　フィードバック・レベルと帰還量

$FL=(D_5-D_3)$	0	1	2	3	4	5	6	7
LEVEL	OFF	$\pi/16$	$\pi/8$	$\pi/4$	$\pi/2$	π	2π	4π

●付表3　位相変調感度

PMS＝D_6～D_4	0	1	2	3	4	5	6	7
MOD. MAX(cent)	0	±5	±10	±20	±50	±100	±400	±700

●付表4　振幅変調感度

AMS	AM MOD(MAX)
0	0
1	23.90625 dB
2	47.8125 dB
3	95.625 dB

●付表5　デチューン（DT1）

OCT	NOTE 上位2ビット	FD＝0 D-CENT	FD＝1	FD＝2	FD＝3	FD＝0 D-FREQ	FD＝1 (Hz)	FD＝2	FD＝3
0	0	0.000	0.000	5.025	10.036	0.000	0.000	0.053	0.107
0	1	0.000	0.000	4.228	8.445	0.000	0.000	0.053	0.107
0	2	0.000	0.000	3.559	7.110	0.000	0.000	0.053	0.107
0	3	0.000	0.000	2.993	5.980	0.000	0.000	0.053	0.107
1	0	0.000	2.515	5.025	5.025	0.000	0.053	0.107	0.107
1	1	0.000	2.115	4.228	6.338	0.000	0.053	0.107	0.160
1	2	0.000	1.778	3.555	5.330	0.000	0.053	0.107	0.160
1	3	0.000	1.496	2.990	4.483	0.000	0.053	0.107	0.160
2	0	0.000	1.258	2.515	5.025	0.000	0.053	0.107	0.213
2	1	0.000	1.057	3.170	4.225	0.000	0.053	0.160	0.213
2	2	0.000	0.889	2.667	3.555	0.000	0.053	0.160	0.213
2	3	0.000	0.748	2.242	3.735	0.000	0.053	0.160	0.267
3	0	0.000	1.258	2.515	3.143	0.000	0.107	0.213	0.267
3	1	0.000	1.057	2.114	3.170	0.000	0.107	0.213	0.320
3	2	0.000	0.889	1.778	2.667	0.000	0.107	0.213	0.320
3	3	0.000	0.748	1.869	2.615	0.000	0.107	0.267	0.373
4	0	0.000	0.629	1.572	2.515	0.000	0.107	0.267	0.427
4	1	0.000	0.793	1.586	2.114	0.000	0.160	0.320	0.427
4	2	0.000	0.667	1.334	2.001	0.000	0.160	0.320	0.480
4	3	0.000	0.561	1.308	1.869	0.000	0.160	0.373	0.533
5	0	0.000	0.629	1.258	1.729	0.000	0.213	0.427	0.587
5	1	0.000	0.529	1.057	1.586	0.000	0.213	0.427	0.640
5	2	0.000	0.445	1.001	1.445	0.000	0.213	0.480	0.693
5	3	0.000	0.467	0.935	1.308	0.000	0.267	0.533	0.747
6	0	0.000	0.393	0.865	1.258	0.000	0.267	0.587	0.853
6	1	0.000	0.397	0.793	1.123	0.000	0.320	0.640	0.907
6	2	0.000	0.334	0.723	1.056	0.000	0.320	0.693	1.013
6	3	0.000	0.327	0.654	0.935	0.000	0.373	0.747	1.067
7	0	0.000	0.315	0.629	0.865	0.000	0.427	0.853	1.173
7	1	0.000	0.315	0.629	0.865	0.000	0.427	0.853	1.173
7	2	0.000	0.315	0.629	0.865	0.000	0.427	0.853	1.173
7	3	0.000	0.315	0.629	0.865	0.000	0.427	0.853	1.173

●付表6　マルチプライ

MUL＝D_3～D_0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
MULTIPLAY	0.5	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

●付表7　キー・スケーリング

KC	KS			
	0	1	2	3
0	0	0	0	0
1	0	0	0	1
2	0	0	1	2
3	0	0	1	3
4	0	1	2	4
5	0	1	2	5
6	0	1	3	6
7	0	1	3	7
8	1	2	4	8
9	1	2	4	9
10	1	2	5	10
11	1	2	5	11
12	1	3	6	12
13	1	3	6	13
14	1	3	7	14
15	1	3	7	15
16	2	4	8	16
17	2	4	8	17
18	2	4	9	18
19	2	4	9	19
20	2	5	10	20
21	2	5	10	21
22	2	5	11	22
23	2	5	11	23
24	3	6	12	24
25	3	6	12	25
26	3	6	13	26
27	3	6	13	27
28	3	7	14	28
29	3	7	14	29
30	3	7	15	30
31	3	7	15	31

＊キー・スケーリングした後のRATEは下式のように，入力レート（R）の2倍に左の表の値（RKs）を加えたもの．

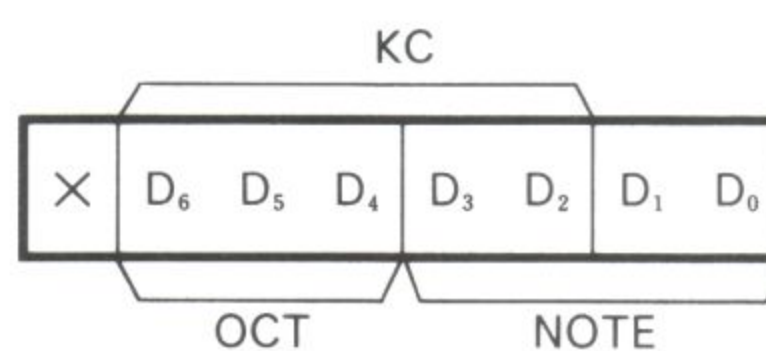
$$RATE = 2 \times R + RKs$$

R：入力の各レート
　AR，D1R，D2Rはレジスタに書き込んだ値を入力レートとし，RRはレジスタに書き込んだ値の2倍に1を加えた値を入力レートとする．
　＊計算結果が63より大きな値のときは，すべてRATE=63

RKs：KEY CODEとKSで定まる値（左表）
　ただし，ここでのKEYCODEは下図のように，NOTEの下位2ビットは切り捨てたKCを用いる．

KC（D_6 D_5 D_4 D_3 D_2）

×	D_6	D_5	D_4	D_3	D_2	D_1	D_0

OCT（D_6 D_5 D_4）　NOTE（D_3 D_2 D_1 D_0）

●付表8　アタック・レート

RATE	msec(10%-90%)	msec(96dB-0dB)	RATE	msec(10%-90%)	msec(96dB-0dB)
153	0.00	0.00	73	35.78	63.72
152	0.24	0.47	72	41.75	74.34
151	0.24	0.47	71	50.10	89.20
150	0.24	0.47	70	62.62	111.51
143	0.30	0.57	63	71.57	127.43
142	0.36	0.67	62	83.50	148.68
141	0.42	0.81	61	100.20	178.41
140	0.59	1.00	60	125.26	223.01
133	0.55	1.09	53	143.15	254.87
132	0.65	1.27	52	167.01	297.35
131	0.78	1.53	51	200.40	356.82
130	0.98	1.91	50	250.50	446.02
123	1.12	1.99	43	286.29	509.74
122	1.31	2.33	42	334.01	594.69
121	1.57	2.78	41	400.81	713.63
120	1.96	3.48	40	501.01	892.04
113	2.24	3.98	33	572.59	1019.48
112	2.61	4.65	32	668.01	1189.38
111	3.13	5.58	31	801.62	1427.27
100	3.91	6.97	30	1002.02	1784.08
103	4.48	7.97	23	1145.16	2038.95
102	5.22	9.29	22	1336.02	2378.78
101	6.27	11.15	21	1603.23	2854.53
100	7.87	13.94	20	2004.04	3568.16
93	8.95	15.93	13	2290.32	4077.90
92	10.44	18.58	12	2672.05	4757.55
91	12.52	22.30	11	3206.45	5709.06
90	15.65	27.88	10	4008.07	7136.33
83	17.89	31.86	03	無限大	無限大
82	20.87	41.53	02	無限大	無限大
81	25.05	44.60	01	無限大	無限大
80	31.32	55.75	00	無限大	無限大

●付表9　ディケイ・レート

RATE	msec(90%-10%)	msec(0dB-96dB)	RATE	msec(90%-10%)	msec(0dB-96dB)
153	1.22	6.02	73	177.43	835.95
152	1.22	6.02	72	207.74	1027.48
151	1.22	6.02	71	249.28	1232.97
150	1.22	6.02	70	311.60	1541.22
143	1.39	8.03	63	356.12	1761.39
142	1.62	8.03	62	415.47	2296.04
141	1.95	9.63	61	498.57	2465.94
140	2.43	12.04	60	623.21	3082.42
133	2.78	13.77	53	712.23	3522.77
132	3.26	16.06	52	830.94	4109.90
131	3.89	19.27	51	997.13	4931.89
130	4.87	24.08	50	1246.41	6164.86
123	5.57	27.52	43	1424.47	7045.55
122	6.49	32.11	42	1661.88	8228.76
121	7.79	38.53	41	1994.26	9863.77
120	9.74	48.16	40	2492.83	12329.71
113	11.12	55.04	33	2848.95	14091.09
112	12.99	64.22	32	3323.77	16439.61
111	15.58	77.06	31	3988.52	19727.54
110	19.48	96.33	30	4985.65	24659.42
103	22.26	110.09	23	5697.88	28182.20
102	25.96	128.43	22	6647.53	32879.23
101	31.16	154.12	21	7977.05	39455.07
100	38.95	192.65	20	9971.30	49318.84
93	44.52	220.17	13	11395.78	56364.40
92	52.83	212.12	12	13295.07	65758.46
91	62.32	308.25	11	15954.08	78910.15
90	77.90	385.31	10	19942.60	98637.69
83	89.03	440.35	03	無限大	無限大
82	103.86	513.74	02	無限大	無限大
81	124.64	616.48	01	無限大	無限大
80	155.80	770.60	00	無限大	無限大

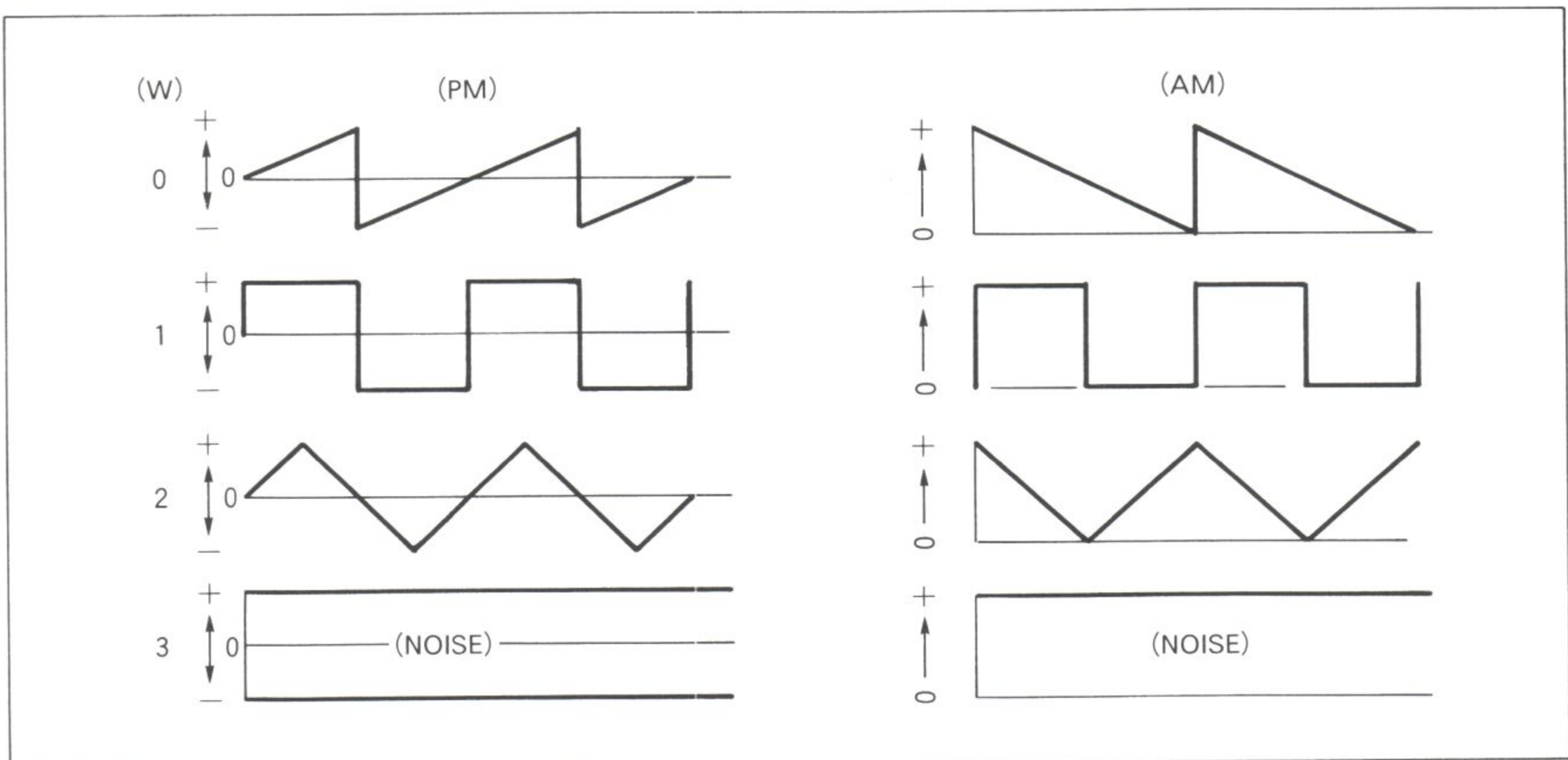
●付表10　デチューン（DT2）

DT2＝D_7～D_6	0	1	2	3
(cent) DETUNE	0	＋600	＋781	＋950
(倍)	1	＋1.41	＋1.57	＋1.73

●付表11　ファースト・ディケイ・レベル

D_7	D_6	D_5	D_4
24	12	6	3

D_7～D_4がすべて“1”＝45dBのときはさらに48dB減衰量が加算される

●付図1　LFO波形コードと発振波形との対応

●付図 2　OCT と NOTE の音程関係（クロック 4 MHz 時）

D 約36.7Hz（OCT 1）　D 約4698.6Hz（OCT 7）

D_6〜D_4	0	1	2	3	4	5	6	7
OCT	0	1	2	3	4	5	6	7

D_3〜D_0	0	1	2	4	5	6	8	9	10	12	13	14
NOTE	D#	E	F	F#	G	G#	A	A#	B	C	C#	D

4-8 音声のサンプリングと合成

音色を PCM 方式で記録するには，一定時間間隔で**サンプリング**して，**AD 変換**によりディジタル化する

●表 4．6　音声合成レジスタ・アドレス・マップ

レジスタ・アドレス	リード/ライト	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	説明
E92001H	READ	REC/PLAY	1	0	0	0	0	0	0	ADPCM ステータス
	WRITE	0	0	0	0	0	REC ST	PLAY ST	SP	ADPCM コマンド
E92003H	READ	B3	B2	B1	B0	B3	B2	B1	B0	入力データ
	WRITE									出力データ
E9A005H	WRITE	IOC7	IOC6	IOC5	IOC4	Sampling RATE		PCM PAN		ADPCM 出力/サンプリング周波数切り換え
E90003H (1BH)	WRITE	CT2	CT 1	×	×	×	×	WAVE FORM		ADPCM クロック切り換え（FM 音源レジスタ・データ・ポート）

●図 4．29　ADPCM のビット構成

(a) ADPCMコマンドのビット構成

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
E92001H	0	0	0	0	0			

D_0：0－ADPCM RECORD/PLAY 動作終了せず　1－ADPCM RECORD/PLAY 動作終了

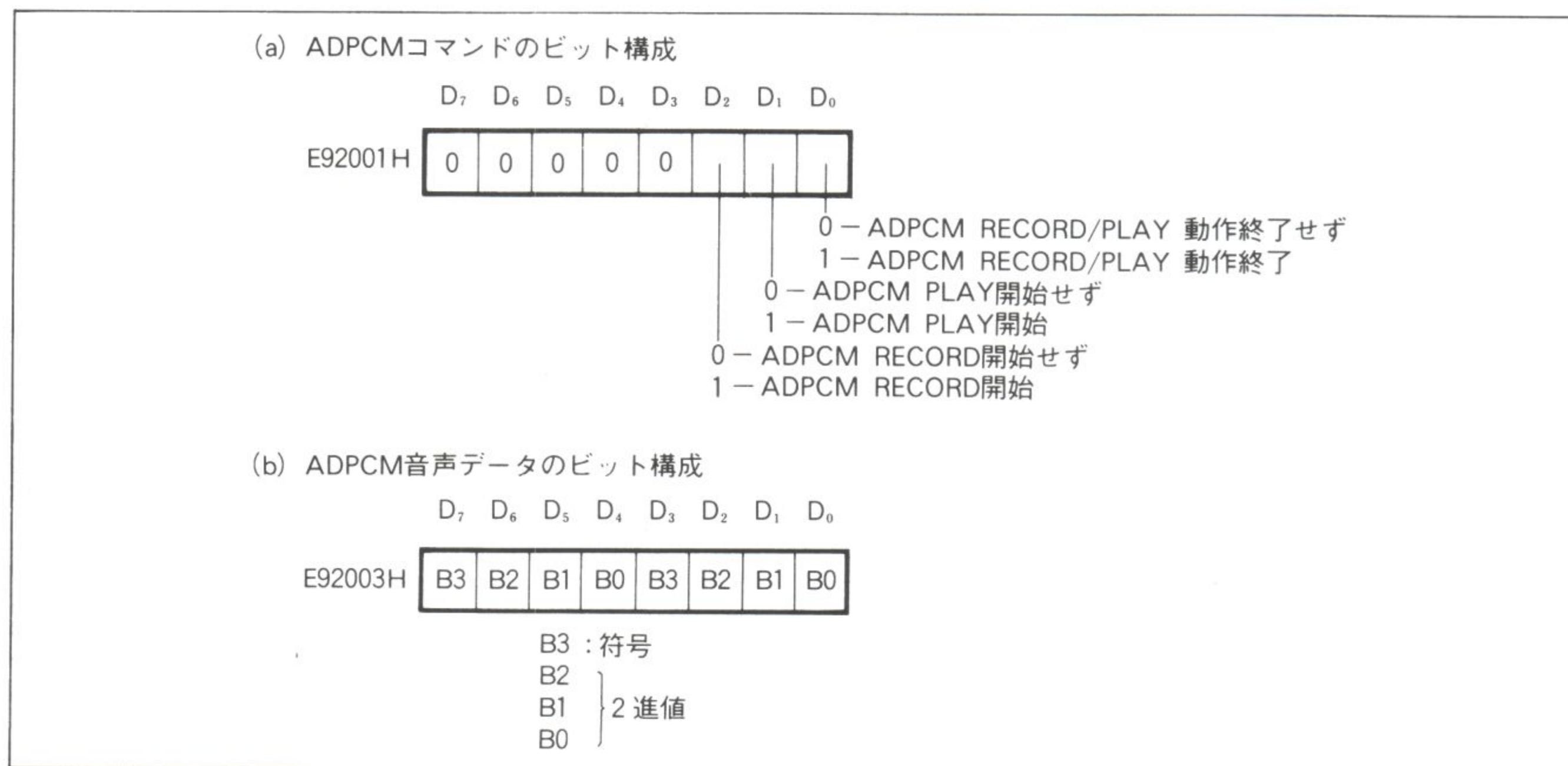
D_1：0－ADPCM PLAY開始せず　1－ADPCM PLAY開始

D_2：0－ADPCM RECORD開始せず　1－ADPCM RECORD開始

(b) ADPCM音声データのビット構成

	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
E92003H	B3	B2	B1	B0	B3	B2	B1	B0

B3 ：符号

B2, B1, B0 ：2 進値

方法をとります．このとき，隣接したデータ値は非常に近く，一般に上位ビットがほとんど同じというパターンが続きます．そこで，１つ前の値との差分をとるようにすれば，個々のデータ値のビット数は少なくてすみます．この理屈を利用したのが，**ADPCM**(Addaptive Differencial PCM)方式です．

サンプリング周波数は3.9，5.2，7.8，10.4，15.6kHzの各値をとることができ，取得されたデータはメモリに蓄積されます．このとき転送速度が比較的速く，データ量も非常に多いため，**DMAC**を用いてCPUが関与しない転送の仕方をするのが普通です．

サンプリング周波数は高いほど周波数帯域が広く，音楽的にもよい音になりますが，限られたメモリ容量では記録時間が少なくなるのが残念です．ただ，人間の話し声などは3.9kHz(電話並み)でも充分聞きとることができ，7.8kHz(ラジオ並み)ではかなりクリアな音になります．だいたいこのあたりが実用的なレベルといえるでしょう．

音声サンプリングと合成に関与するレジスタとアドレスの一覧は，表4.6のとおりです．このうち，FM音源のデータ・ポートのうちCT1だけが原発振クロック切り換え用として供給されているので注意が必要です．ほかには，アドレスE92001H，E92003HがそれぞれMSM6258(ADPCM用のIC)，アドレスE92005Hが8255に割り当てられています．

MSM6258のコマンドは図4.29のように簡単なもので，同じアドレスから読み出すとステータス(図4.30)が参照できます．また，8255による環境の設定は図4.31のとおりです．言うまでもありませんが，コマンドは環境設定がすんでから出さなければなりません．

8255のＣポート上位４ビットは，ジョイスティック制御に用いられます．PCMPANの制御は，D_0，D_1

●図４．30　ADPCMステータスのビット構成

	D7	D6	D5	D4	D3	D2	D1	D0
E92001H		1	0	0	0	0	0	0

D7:
0－ADPCM PLAY中
1－ADPCM RECORD中

●図４．31　8255による環境の設定

	D7	D6	D5	D4	D3	D2	D1	D0
E9A005H								

D1 D0:
0　0－ADPCM音声出力Cut
0　1－ADPCM音声出力(Left) Out
1　0－ADPCM音声出力(Right) Out
1　1－ADPCM音声出力(Left&Right) Out

D3 D2:
(４MHz/８MHz)
0　0－ ADPCMサンプリング周波数3.9KHz/7.8KHz
0　1－ ADPCMサンプリング周波数5.2KHz/10.4KHz
1　0－ ADPCMサンプリング周波数7.8KHz/15.6KHz
1　1－ 使用せず(禁止)

D4:
0－ジョイスティックNo.１通常動作
1－No.１無効

D5:
0－ジョイスティックNo.２通常動作
1－No.２無効

D6:
0－ジョイスティックNo.１オプションＡ
1－オプション機能(信号名IOA５)

D7:
0－ジョイスティックNo.１オプションＢ
1－オプション機能(信号名IOA６)

がそれぞれL，Rチャンネルのスイッチになっており，1のとき該当チャンネルに出力できます．両方とも0ならば，たとえMSM6258が動作状態にあっても，音声ラインには出力されません．

FM音源の設定は図4.32のとおりです．

●図4.32　FM音源の設定（PCM関係）

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

E92003H
(1BH)

WAVE FORM

CT1　音声合成クロック切り換えポート（原発振周波数切り換え）
　0：8MHz
　1：4MHz

CT2　FDC強制READYポート
　0：FDDのREADY状態（通常動作）
　1：強制READY

4-9 ディスク

FDの仕様は，表4.7のように2HDを中心に2DD，2Dまで対応できるようになっています．内蔵FDドライブの仕様は表4.8のとおりで，2HD以外に対応するには外付けドライブが必要になります．

X68000のディスク関係の周辺ブロック図を，図4.33に示します．

ここでFDCにはμPD72065を使用し，アドレスは表4.9のように配分されています．このうちFDC関係のレジスタは，ステータス・レジスタ（表4.10）とデータ・レジスタで，データ・レジスタはそのときの状態（フェーズ）によりコマンドを受け付けたり，読み書きする入出力データを中継する（入れ物）として使われます．

ドライブ・ステータス（図4.34）とドライブ・コントロール（図4.35）は，X68000独自の信号で，イジェクトなどの制御に使います．

また，アクセス・ドライブ・セレクトなど（図4.36）のポートは，2HD/2D（2DDを含む）の切り換えもします．

割り込み関係では，参照するステータス（図4.37）と，マスク用ポート（図4.38），ベクトル書き込み用のポート（図4.39）をもっています．

一般的なディスクのアクセスは，ドライブを決め，シーク（目的トラックまで移動）をして，その後読み書きをするという手順で行なわれます．シークまでと読み書きの動作を開始させるまではCPUが関与しますが，データの転送はきわめて高速なため，普通DMACによってカバーします．またCPUによる直接転送が可能であっても，転送中の割り込みに対応できなくなるという問題が生じ，FD入出力中にキーボードからのデータが無視されるといった状態も起こります．このため，CPUによる直接転送は，コストが安い

●表4.7　＃1000におけるディスク仕様

128, 256, 512, 1,024Bytes/sector 26, 16, 8Sector/track 77Track/side 154Track/disk (Format時1.28MBytes)	FDC 入力クロック 2HD (8MHz) 2DD・2D (4MHz) 切り換え可能 ただし，2DD・2DタイプのFDDを接続する場合は，拡張用のフロッピーディスク・コネクタを使用．	変調方式 MFM FM 切り換え可能

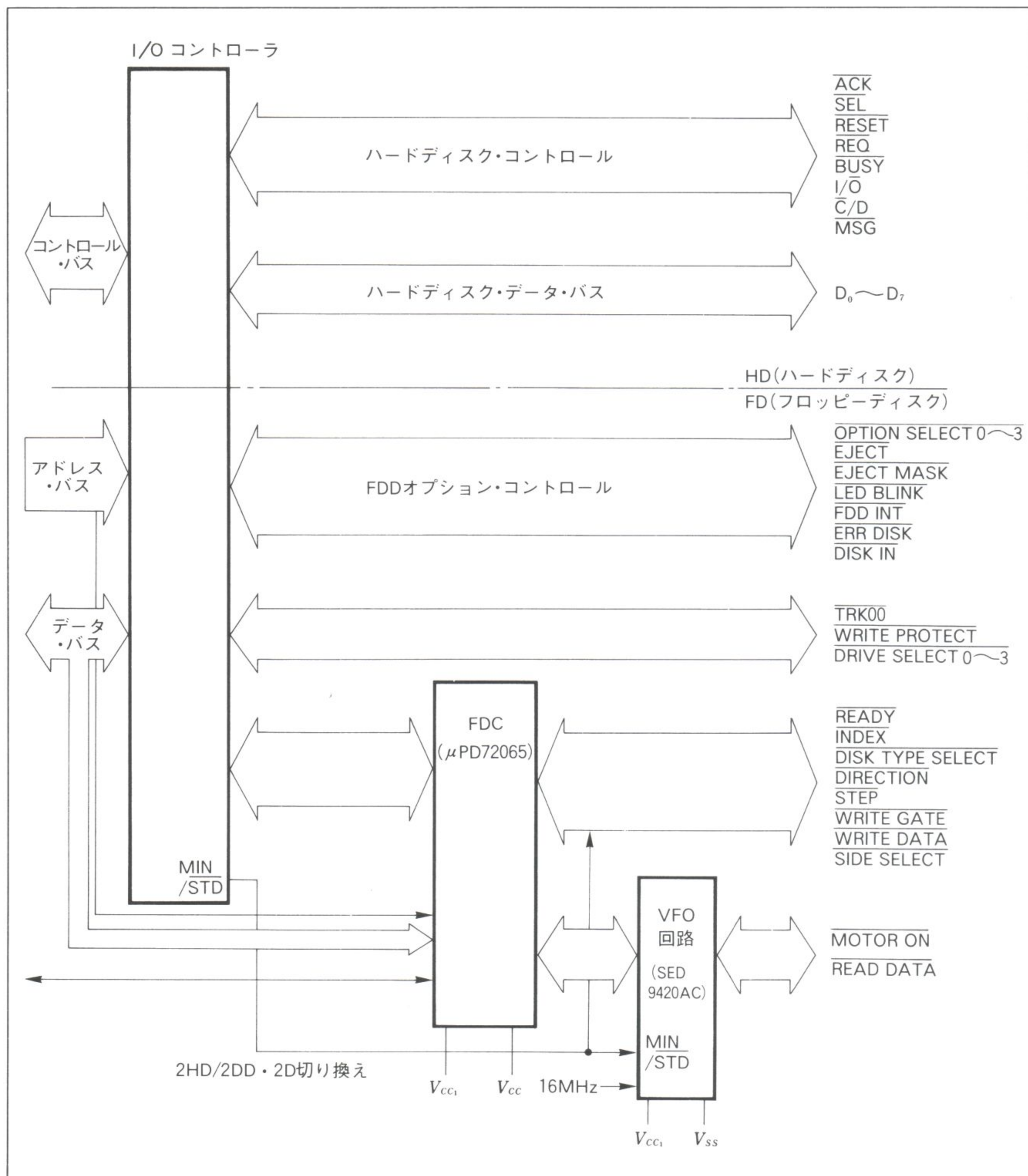
*）なお，拡張用のフロッピーディスク・コネクタを使用して，X1，X1turbo用の2D・2DDタイプFDDを接続する場合は，その接続ケーブルの中の20番ピン～29番ピンは，無接続にすること（＃1000で，オプションに使用）．

●表4．8　内蔵 FDD 仕様

項目		値	備考
記憶容量 (KBytes)	アンフォーマット時	1,667	
	フォーマット時	1,065	IBM 準拠26セクター，256バイト/高密度モード
	トラック容量	10.42	
データ転送速度（K bits/sec）		500	
アクセス・タイム (msec)	トラック間移動時間	3	シーク時の待ち時間＝トラック間移動時間＋シーク・セトリング時間 平均アクセス時間＝平均トラック移動時間＋シーク・セトリング時間
	シーク・セトリング時間	15	
	平均アクセス時間	95	
メディア回転数（rpm）		360	
スピンドル・モーター起動時間（sec）		0.5	
最内周記録密度（BPI）		9,870	
トラック数	TRACK/SIDE	77	
	TRACK/DISK	154	
トラック密度（TPI）		96	
ヘッド数		2	
変調方式		MFM	FM方式も可能
外形寸法（mm）		27.0(H)＊148.0(W)＊198.0(D)	
重量		900 g	
その他		オートクランプ・イジェクト機構，オートリキャリー・ブレート機能，VFO（SED9420AC）	

●表4．9　FD 系統のアドレス配分

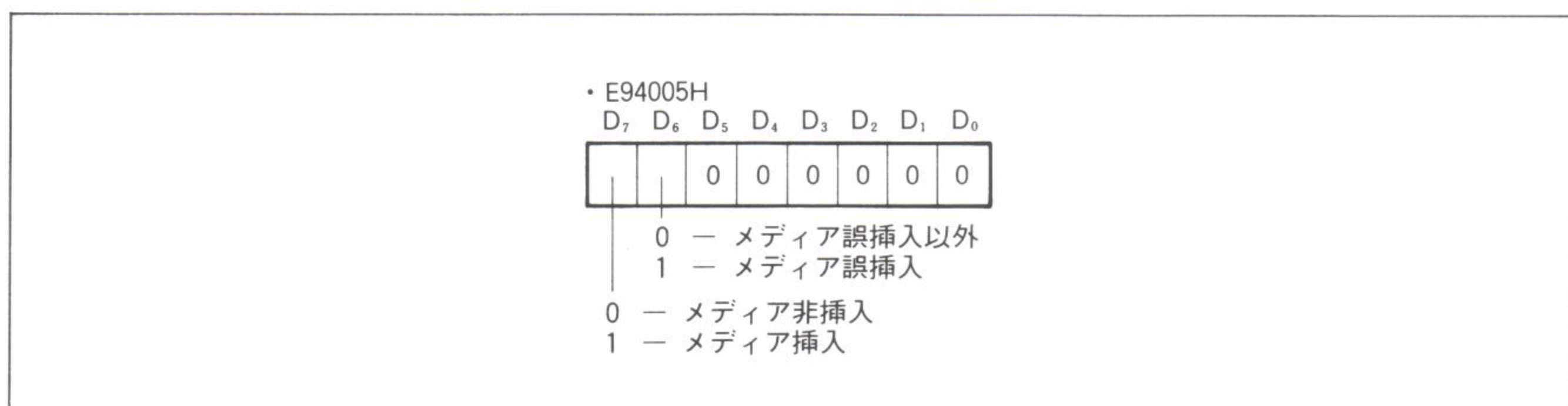
アドレス	リード/ライト	レジスタ，I/O ポート
E 94001 H	Read	FDC ステータス・レジスタ
	Write	
E 94003 H	Read	FDC データ・レジスタ
	Write	FDC データ・レジスタ
E 94005 H	Read	ドライブ・ステータス（オプション信号）
	Write	ドライブ・コントロール（オプション信号）
E 94007 H	Read	
	Write	アクセス・ドライブ・セレクト，2HD/2DD・2D切り換え
E 9C001 H	Read	割り込み信号ステータス
	Write	割り込み信号マスク
E 9C003 H	Read	
	Write	割り込みベクタ番号

●図4．33　FDD周辺ブロック図

●図4．34　ドライブ・ステータス（In）――オプション信号

・E94005H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
		0	0	0	0	0	0

D_6:
0 ― メディア誤挿入以外
1 ― メディア誤挿入

D_7:
0 ― メディア非挿入
1 ― メディア挿入

●表4．10　ステータス・レジスタの内容

ビット	名　称	略称	内　容
D_7	Request for Master	RQM	メイン・システムに対してデータをやりとりする準備ができていることを示す． DIO（D_6ビット）の状態により，次のように動作する． D_{10}＝0 のとき： メイン・システムが FDC へデータを送る場合で，メイン・システムが FDC にデータを書き込むと RQM＝0 となり，FDC がそのデータを引き取ると RQM＝1 になる． • C-Phase，コマンド待ち • Non-DMA ライトの E-Phase • SEEK系の E-Phase D_{10}＝1 のとき： FDC がメイン・システムへデータを送る場合で，FDC がデータ・レジスタにデータをセットすると RQM＝1 となり，メイン・システムがそのデータを読み取ると RQM＝0 になる． • R-Phase • Non-DMA リードの E-Phase（READ ID を除く）
D_6	Data Input/Output	DIO	メイン・システムと FDC の間でやりとりするデータの方向を示す．0のときはメイン・システムから FDC の方向，1のときは FDC からメイン・システムの方向．
D_5	Non-DMA Mode	NDM	Non-DMA モードでデータ転送中（E-Phase）であることを示す． C-Phase，R-Phase ではこのビットはリセットされている．
D_4	FDC Busy	CB	C-Phase，R-Phase またはリード/ライト・コマンドの E-Phase であることを示す（SEEK系の E-Phase ではセットしない）． このビットをセットしているときは次のコマンドを受け付けない．
D_3	FD3 Busy	D3B	デバイス＃3にシーク動作をさせているか，またはシーク動作終了の割り込みを保留中であることを示す（E-Phase）．このビットがセットされているとき，リード/ライト系のコマンドは書き込み禁止．
D_2	FD2 Busy	D2B	デバイス＃2について D_3ビットと同内容
D_1	FD1 Busy	D1B	デバイス＃1について D_3ビットと同内容
D_0	FD0 Busy	D0B	デバイス＃0について D_3ビットと同内容

●図4．35　ドライブ・コントロール（Out）――オプション信号

●図4．36　アクセス・ドライブ・セレクト，2HD／2DD・2D切り換え（Out）

・E94007H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
	0	0		0	0		

D_1 D_0:
0 0 ― アクセス・ドライブ0指定
0 1 ― アクセス・ドライブ1指定
1 0 ― アクセス・ドライブ2指定
1 1 ― アクセス・ドライブ3指定

D_4:
0 ― 2HD（1，6Mタイプ）
1 ― 2DDまたは，2D（1Mまたは，500Kタイプ）

D_7:
0 ― ドライブ・セレクト・ディセーブル，モーターOFF
1 ― ドライブ・セレクト・イネーブル，モーターON

●図4．37　割り込み信号ステータス（In）

E9C001H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0

D_0:
0 ― プリンタ割り込みマスク中
1 ― プリンタ割り込みマスク解除中

D_1:
0 ― FDD割り込みマスク中
1 ― FDD割り込みマスク解除中

D_2:
0 ― FDC割り込みマスク中
1 ― FDC割り込みマスク解除中

D_3:
0 ― ハードディスク割り込みマスク中
1 ― ハードディスク割り込みマスク解除中

D_4:
0 ―ハードディスク割り込みOFF―― コマンド終了時以外
1 ―ハードディスク割り込みON ―― コマンド終了時

D_5:
0 ―プリンタ割り込みOFF(BUSY＝1)―― プリンタへデータ送信不可
1 ―プリンタ割り込みON（BUSY＝0)―― プリンタへデータ送信可

D_6:
1 ― FDD割り込みON ―― メディア挿入時(メディアクランプ完了後，割り込み)または，
メディア排出時(メディアのクランプがはずれ，排出時に割り込み)
0 ― FDC割り込みOFF ―― 上記以外

D_7:
1 ― FDD割り込みON ―― リード／ライトのE-PHASE終了によるリザルトステータスの読み取り要求または，SEEK，RECALIBRATEコマンドのE-PHASE終了によるSENSE INTERRUPT STATUSコマンドの起動要求，または，デバイスの状態遷移によるSENSE INTERRUPT STATUSコマンドの起動要求
0 ― FDC割り込みOFF ――上記以外

●図4．38　割り込み信号マスク（Out）

・E9C001H

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0				

D_0:
0 ― プリンタ割り込みマスク
1 ― プリンタ割り込みマスク解除

D_1:
0 ― FDD割り込みマスク
1 ― FDD割り込みマスク解除

D_2:
0 ― FDC割り込みマスク
1 ― FDC割り込みマスク解除

D_3:
0 ― ハードディスク割り込みマスク
1 ― ハードディスク割り込みマスク解除

●図4．39 割り込みベクタ番号（Out）

D_7(DISK IN)	D_6 (ERROR DISK)	機　能
1	1	メディア誤挿入（この場合，メディア挿入時と排出時に割り込みがかかる）
1	0	メディア挿入
0	0	メディア非挿入

にもかかわらず嫌われているのが実態です．

μPD72065についての詳しいことは，マニュアルを読んでいただくとして，ここでは要点だけかいつまんで説明します．

このFDCで使えるコマンドの種類は，表4.11のとおりで，一般には**シーク，リード・データ，ライト・データ**が頻繁に使われます．通常の入出力が，セクタ単位で行なわれるため，このことはきわめて当然なのですが，例外的に**フォーマット**時にはライトIDによって1トラック分の内容を一度に書き込むことも行なわれます．反対に一度に読み込むには**リード・ダイアグノステック**（診断読み）によって実行でき，このコマンドを使うとマルチ・セクタなどエラーのあるセクタの読み出しもできます．

●表4．11 FDCのコマンド

分　類	コマンド名	概　略
リード系	READ DATA	セクタを指定してそのデータをホストへ転送する．
	READ DELETED DATA	
	READ ID	1セクタ分のIDを読み出す．
	READ DIAGNOSTIC	トラックのフォーマットをチェックする．
	SCAN EQUAL	1セクタごとにデータをホストのデータと比較し，条件に合うセクタを検出する．
	SCAN LOW OR EQUAL	
	SCAN HIGH OR EQUAL	
ライト系	WRITE DATA	セクタを指定してホストのデータを転送する．
	WRITE DELETED DATA	
	WRITE ID	1トラック分のトラック・フォーマットを書き込む．
シーク系	RECALIBRATE	リード/ライト・ヘッドを最外トラック（トラック0）へ移動させる．
	SEEK	リード/ライト・ヘッドを指定トラックへ移動させる．
センス系	SENSE INTERRUPT STATUS	FDC内部の割り込み要因（シーク・エンド，状態遷移）を読み出す．
	SENSE DEVICE STATUS	FDDの状態を読み出す．
イニシャライズ	SOFTWARE RESET	FDDを初期状態とする．
	SPECIFY	FDCの動作モードを定義する．
スタンバイ制御	SET STANDBY	FDCをスタンバイ状態とする．
	RESET STANDBY	FDCのスタンバイ状態を解除する．

●図4．40　リード・データのデータ・レジスタ・フォーマット

フェーズ		MSB							LSB	備考
	コマンド1	MT	MF	SK	0	0	1	1	0	
	コマンド2	×	×	×	×	×	HD	US1	US0	
	パラメータ1	←C→								実行開始セクタ
	パラメータ2	←H→								実行開始セクタ
(Cフェーズ)	パラメータ3	←R→								実行開始セクタ
CPU→FDC	パラメータ4	←N→								実行開始セクタ
	パラメータ5	←EOT→								
	パラメータ6	←GSL→								
	パラメータ7	←DTL→								
(Eフェーズ)	データ転送（フロッピー→メモリ）									
	ST0	←IC→		SE	EC	NR	HD	US1	US0	D_7，D_5，D_4ビットは常に0
	ST1	EN	0	DE	OR	0	ND	NW	MA	D_6，D_3，D_1ビットは常に0
	ST2	0	CM	DD	NC	SH	SN	BC	MD	D_7，D_3，D_2ビットは常に0
(Rフェーズ)	パラメータ1	←C→								
FDC→CPU	パラメータ2	←H→								
	パラメータ3	←R→								
	パラメータ4	←N→								

Rフェーズ パラメータ1～4：
IC(ST0)＝00：実行終了セクタの次のセクタ
IC(ST0)＝01：実行終了（異常発生）セクタ
ただし，IC(ST0)＝00，CM(ST2)＝1で，かつSK(コマンド1)＝0のときは実行終了セクタ

●図4．41　ライト・データのデータ・レジスタ・フォーマット

フェーズ		MSB							LSB	備考
	コマンド1	MT	MF	SK	0	0	1	0	1	
	コマンド2	×	×	×	×	×	HD	US1	US0	
	パラメータ1	←C→								実行開始セクタ
	パラメータ2	←H→								実行開始セクタ
(Cフェーズ)	パラメータ3	←R→								実行開始セクタ
CPU→FDC	パラメータ4	←N→								実行開始セクタ
	パラメータ5	←EOT→								
	パラメータ6	←GSL→								
	パラメータ7	←DTL→								
(Eフェーズ)	データ転送（メモリ→フロッピー）									
	ST0	←IC→		SE	EC	NR	HD	US1	US0	D_7，D_5ビットは常に0
	ST1	EN	0	DE	OR	0	ND	NW	MA	D_6，D_3ビットは常に0
	ST2	0	CM	DD	NC	SH	SN	BC	MD	D_7，D_6，D_5，D_3，D_2，D_0ビットは常に0
(Rフェーズ)	パラメータ1	←C→								
FDC→CPU	パラメータ2	←H→								
	パラメータ3	←R→								
	パラメータ4	←N→								

Rフェーズ パラメータ1～4：
IC(ST0)＝00：実行終了セクタの次のセクタ
IC(ST0)＝01：実行終了（異常発生）セクタ

コマンドの実行に際しては，コマンドの転送(**Cフェーズ**)，実行およびデータの転送(**Eフェーズ**)，実行結果の確認(**Rフェーズ**)の各段階でFDCの動作状態が変わります．たとえばリード・データでは，データ・レジスタが各段階ごとに図4.40のようなフォーマットで使用されます．またライト・データについては，図4.41のフォーマットになります．

なお，関連LEDの点灯関係は表4.12のとおりです．

●表4．12　FD関連LEDの点灯関係

フロント POWERスイッチ	アクティビティLED	イジェクトLED
ON状態	•メディアがFDDに入っている場合緑色が点灯 •メディアがFDDに入っていないときで，かつLED点滅機能がONの場合緑色が点滅 •メディアがFDDに入っていないときでかつLED点滅機能がOFFの場合消灯 ↓ •FDDをリード/ライトする場合（ドライブ・セレクトON，レディON）緑色から赤色点灯に変化	•イジェクト・スイッチ・マスク機能がONの場合消灯 •メディアがFDDに入っているときでかつイジェクト・スイッチ・マスク機能がOFFの場合緑色が点灯
OFF状態	•メディアがFDDに入っている場合緑色が点灯 •メディアがFDDに入っていない場合消灯	•メディアがFDDに入っている場合緑色が点灯 •メディアがFDDに入っていない場合消灯

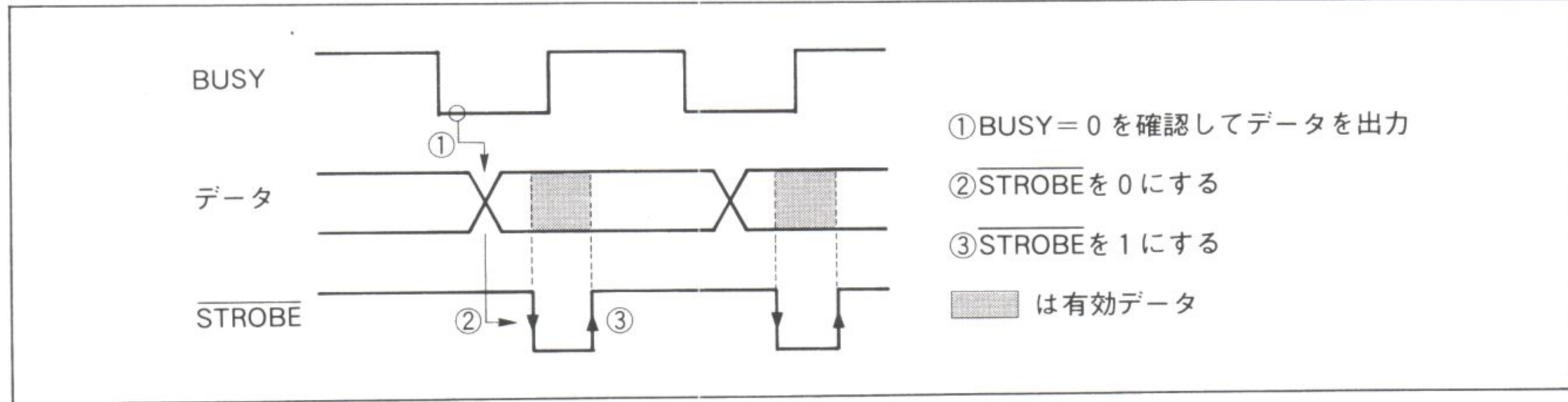
4-10 プリンタ

ほとんどのパソコンで採用しているプリンタのインターフェース規格は，セントロニクス方式による8ビット並列転送です．X68000でもその例にもれずセントロニクス方式を採用しています．

この方式の転送手順を，図4.42に示します．ここではBUSY信号を監視し，0の状態を転送可能とみなしてデータを送出し，$\overline{\text{STROBE}}$を0にすることによりデータ送出中であることをプリンタに通知します．そして$\overline{\text{STROBE}}$が0になっている間は，データは決して変更しないようにします．受信側では，$\overline{\text{STROBE}}$の立ち上がりでデータの取得を中止する仕組にしておけば，その後のデータ値の変動の影響を受けることがありません．

このような動作をさせるために，X68000では図4.43のレジスタ構成をとっています．ここで，プリンタ・データ，ストローブは上述の説明に対応し，BUSY信号を監視するために割り込みを利用する方法をとっています．こうすれば，プログラムでループして調べる必要はありません．このための割り込みを許すかどうかは，アドレスE9C001Hにある割り込み制御レジスタD_0の設定で決められ，割り込みが生じたことの確認はD_4で確認することができます．このほかに，E9C003HにプリンタBUSY割り込み時のベクトル

●図4．42　セントロニクス方式によるデータ転送

BUSY
データ
$\overline{\text{STROBE}}$
①BUSY＝0を確認してデータを出力
②$\overline{\text{STROBE}}$を0にする
③$\overline{\text{STROBE}}$を1にする
は有効データ

●図4．43　プリンタ・レジスタ・マップ

レジスタ・アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備考
E8C001H	プリンタ出力データ								プリンタ・データ・レジスタ（ライト動作のみ）
E8C003H								STR0	プリンタ・ストローブ・レジスタ（ライト動作のみ）
E9C001H (Write)									
(Read)									
E9C003H	割り込みベクタ						1	1	プリンタ・ビジー割り込みベクタ

E9C001H (Write)

- D_0：0 — プリンタ・ビジー割り込みマスク／1 — プリンタ・ビジー割り込みマスク解徐
- D_1：0 — FDD割り込みマスク／1 — FDD割り込みマスク解除
- D_2：0 — FDC割り込みマスク／1 — FDC割り込みマスク解除
- D_3：0 — ハードディスク割り込みマスク／1 — ハードディスク割り込みマスク解除

(Read)

- D_0：0 — プリンタ・ビジー割り込みマスク中／1 — プリンタ・ビジー割り込みマスク解除中
- D_1：0 — FDD割り込みマスク中／1 — FDD割り込みマスク解除中
- D_2：0 — FDC割り込みマスク中／1 — FDC割り込みマスク解除中
- D_3：0 — ハードディスク割り込みマスク中／1 — ハードディスク割り込みマスク解除中
- D_4：0 — ハードディスク割り込みOFF／1 — ハードディスク割り込みON
- D_5：0 — プリンタ・ビジー割り込みOFF（BUSY=1）――プリンタへデータを送信不可／1 — プリンタ・ビジー割り込みON（BUSY=0）――プリンタへデータを送信可
- D_6：0 — FDD割り込みOFF／1 FDD割り込みON
- D_7：0 — FDC割り込みOFF／1 — FDC割り込みON

値を事前に設定することが必要です．

$\overline{\text{STROBE}}$の制御は，プログラムによりD_0の値を1にするか0にするかということだけで，この値が直接プリンタに転送されます．

4-11　ジョイスティック

ゲームの世界にも標準規格があり，ジョイスティックは**アタリ社**の規格に合ったものがよく使われています．また，一般的にパソコンでは，オプション扱いとなっている点も共通していて，ジョイスティックは別途購入しなければならないことになっています．しかし，共通規格というのは便利なもので，たとえばX68000にFMシリーズのジョイスティックをつないでも問題なく動きます．

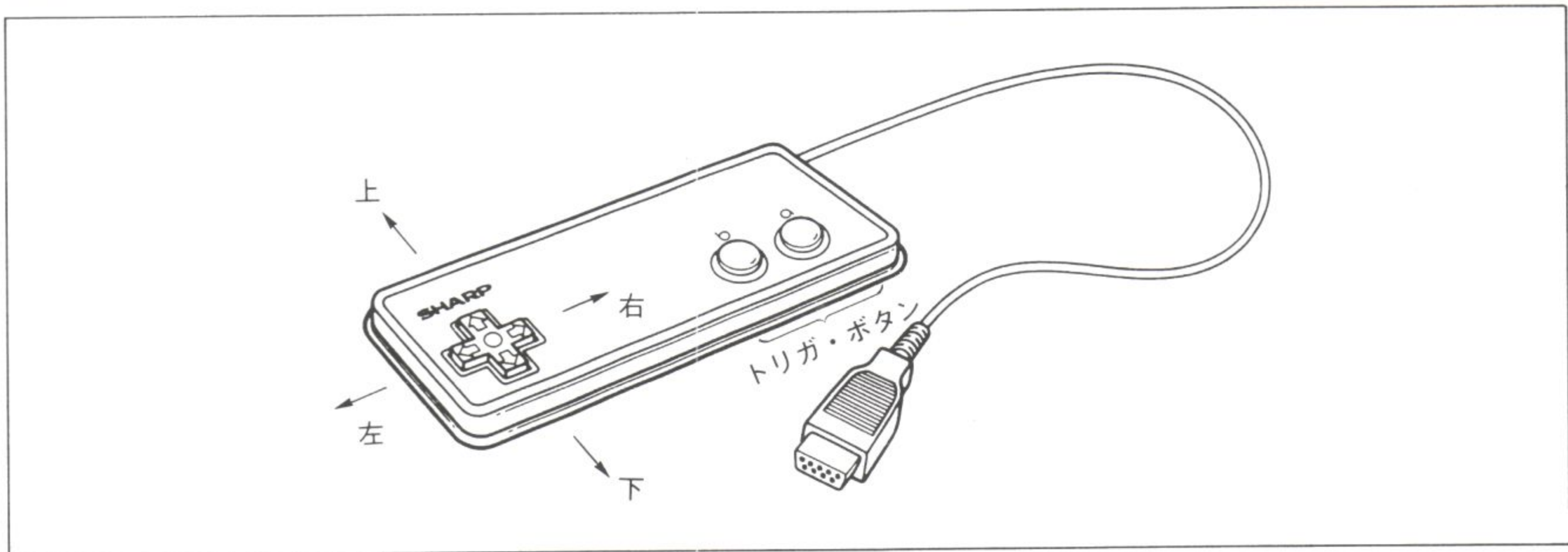
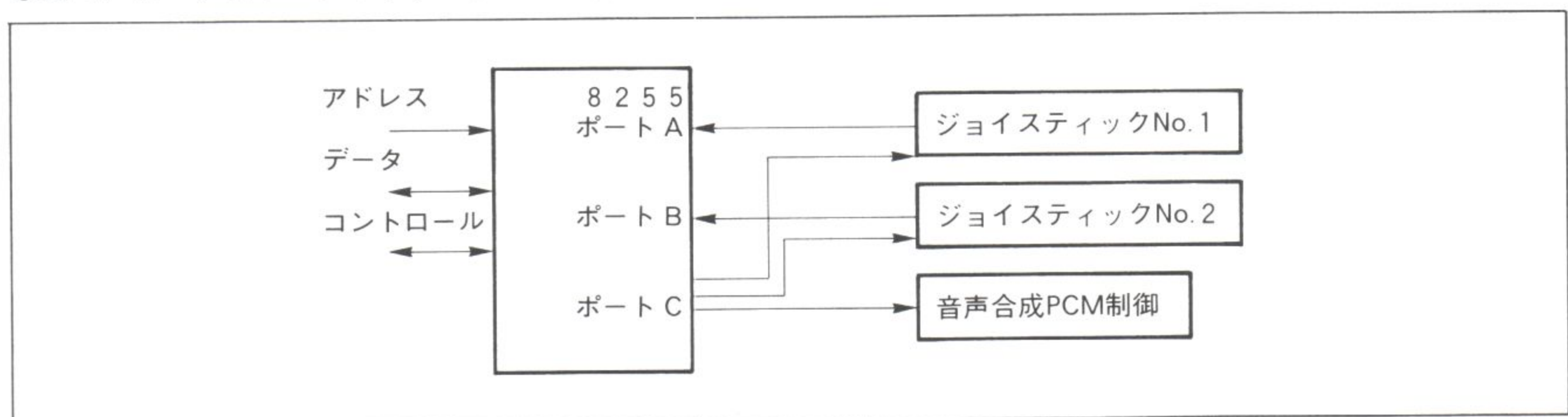
ジョイスティックの原理はきわめて簡単なもので，図4.44に関連づけて説明すると，左右上下方向に対応する**スイッチ**と，**トリガ・ボタンA・B**との合計6個のスイッチが内蔵されているだけです．方向についての分解能は45°単位で，たとえば右上は右と上のスイッチがONになることで識別します．したがって，上と下など，反対方向にあるスイッチが同時にONになることはあり得ず，どれか1つか，隣接した2個のスイッチがONになるケースしか存在しないのがジョイスティックの特徴です．

ジョイスティックの制御などのブロック図は図4.45のとおりで，8255のCポートから音声合成制御のた

●図4．44　ジョイスティックの概観

上
右
左
下
SHARP
トリガ・ボタン

●図4．45　ジョイスティック・ブロック図

アドレス
データ
コントロール
8255
ポートA
ポートB
ポートC
ジョイスティックNo.1
ジョイスティックNo.2
音声合成PCM制御

●図4．46　ジョイスティック・レジスタ・アドレス・マップ

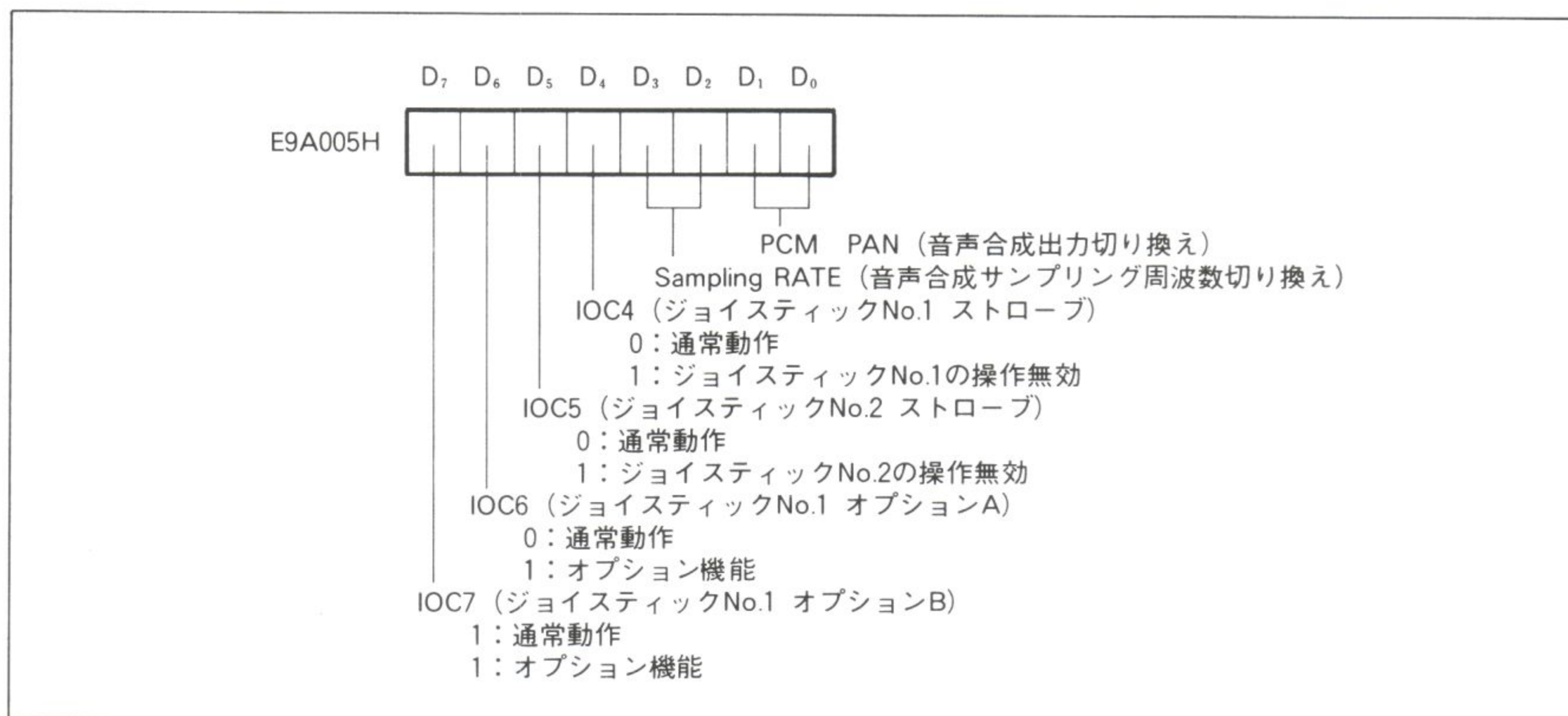
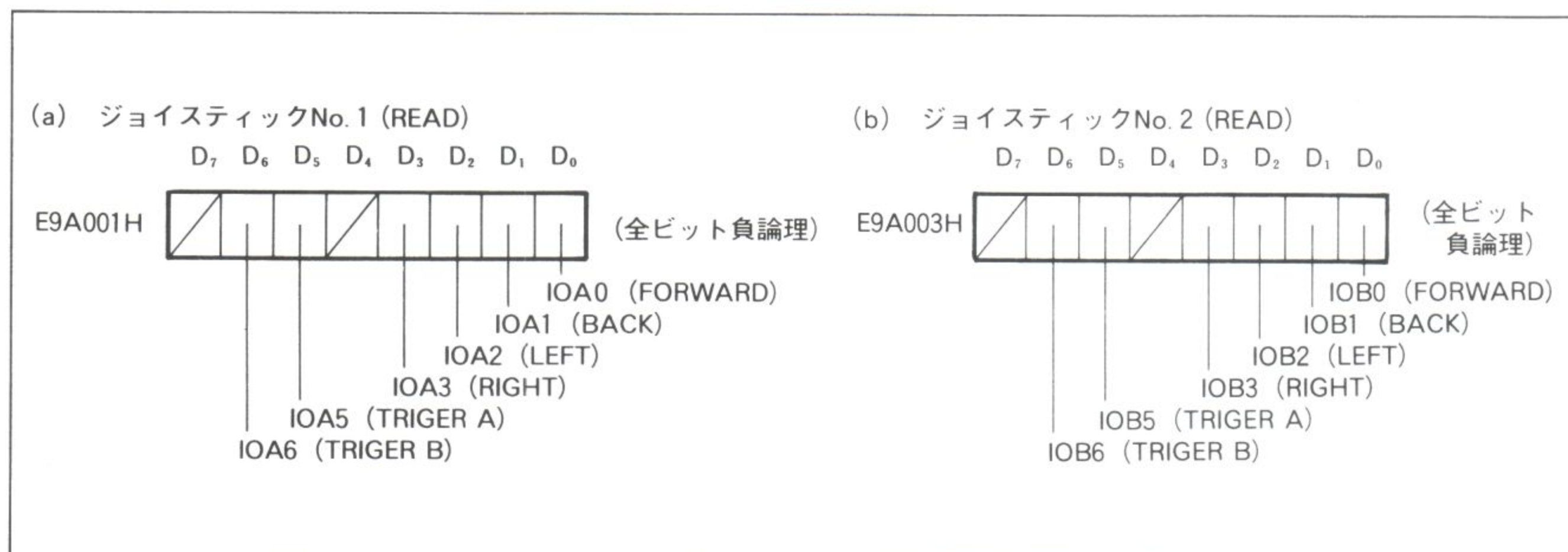
レジスタ・アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
E9A001H	／	←		／	ジョイスティックNo.1			
E9A003H	／	←		／	ジョイスティックNo.2			
E9A005H	IOC7	IOC6	IOC5	IOC4	Sampling Late		PCM　PAN	

めのビットとともにジョイスティック制御線が出されています．また，ジョイスティックからのスイッチ情報は，8255のA，Bポートから読み取るようになっていて，これらのアドレス関係は図4.46のように設計されています．

8255からのジョイスティック・コントロールは，図4.47のようなビット構成をとっているため，標準的な動作をさせるためにはD_4～D_7の全ビットを0に設定することが必要です．

データの取得は8255のA，Bポートから読み取る（図4.48）だけですが，スイッチがOFFのときは1，ONのときは0になっている点に注意が必要です．これは，ジョイスティックが接続されていないとき常に1になるように設計されているためで，このようにすることで各スイッチの共通線をグランド（0レベル）にできるという電気的な配慮がなされています．

マウスの場合はかなり頻繁なアクセスが必要ですが，ジョイスティックでは単なるスイッチのON/OFFのため，あまり頻繁に読み取って座標情報に反映すると，あっという間にスクリーンからはみ出してしまいます．したがって，読み出し間隔は一般的に長いほうがよいのですが，あまり長すぎると操作者がイライラすることになりかねないので，最適間隔は実験しながら調整するのがよいでしょう．

●図4．47　ジョイスティック・コントロール

●図4．48　ジョイスティック・レジスタ詳細

4-12　X68000でのDMACの使われ方

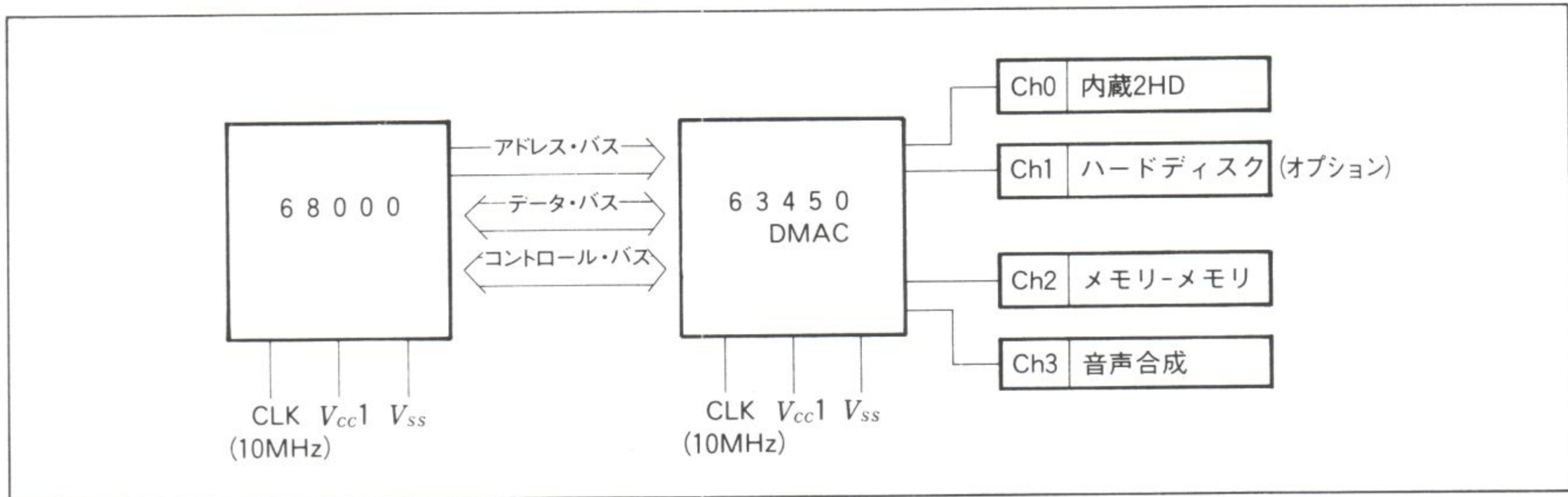
DMACは，CPUに代わってメモリ，周辺装置間のデータ転送を行なうデバイスで，単純大量アクセスを肩代わりすることによりCPUの負担を軽くし，割り込みなどに対する応答性を高める働きがあります．

X68000に使われているDMACは，68450のC-MOS版，63450です．このICは68000用に設計され，4つの**チャンネル**をもっています．ここでいうチャンネルとは，DMA(ダイレクト・メモリ・アクセス：CPUに代わってメモリなどを直接アクセスする)機能のワン・セットを言い，並行して4つのデバイスとのDMA

●表4．13　DMAC各チャンネル割り当て

チャンネルNo.	割り当て	転送要求	ブロック転送
0	内蔵2HD	外部転送要求 サイクル・スチール・モード	・全チャンネル デュアル・アドレス・モード ・全チャンネル(プログラマブル) コンティニュアス・モード アレイ・チェイン・モード リンク・アレイ・チェイン・モード
1	ハードディスク(オプション)	同　上	
2	メモリ-メモリ	オート・リクエスト 限定速度，最大速度	
3	音声合成	外部転送要求 サイクル・スチール・モード	

●図4．49　DMACブロック図

68000
アドレス・バス
データ・バス
コントロール・バス
63450 DMAC
Ch0　内蔵2HD
Ch1　ハードディスク（オプション）
Ch2　メモリ-メモリ
Ch3　音声合成
CLK　$V_{CC}1$　V_{SS}
（10MHz）
CLK　$V_{CC}1$　V_{SS}
（10MHz）

が行なえることを意味します．X68000では，各チャンネルを表4.13のように割り付けています．したがってCPUも含めたハードウェア・ブロックの関連は図4.49のように描けます．

DMACは一般的に，外部からの転送要求線の信号が発生するたびに1回または複数回の転送動作を繰り返すために使いますが，転送要求線をもっていないデバイスに対しては，自動的に連続して転送動作を繰り返す**オート・リクエスト**機能で代替することができます．これらの詳細については，表4.14を参照してください．

またブロック転送の基本的な方法は，転送開始アドレスとワード数をDMACにセットしてすべてを任せる（**単一データ・ブロック転送**）もので，この方法が制御も簡単なためよく使われます．しかし，メモリ領域

●表4．14　DMAC転送要求方法

転送要求方法			
外部転送要求（$\overline{REQ}$ピンを用いる）	サイクル・スチール・モード	単一オペランドごとに転送する．各転送終了のつどバスを放す．
	サイクル・スチール・バスホールド・モード	単一オペランドごとに転送する．ただし，転送後一定期間バスを放さない．
	バースト・モード	複数オペランドを連続して転送する．
オート・リクエスト（$\overline{REQ}$ピンを用いない）	限定速度	複数オペランドを連続して転送する．ただし，途中でバスを定期的に放し，バスを占有しない．
	最大速度	複数オペランドを連続して転送する．ただし，途中でバスを放さず，転送終了までバスを占有する．
オート・リクエスト（最初のオペランド転送のみ）＋外部転送要求（2番目以降のオペランド転送）		

●表4．15　DMACデータ・ブロック転送

		〈手続き〉
単一データ・ブロック転送		DMAC内部レジスタに転送アドレスと転送語数を与える．
複数データ・ブロック転送	コンティニュー・モード	DMAC内部レジスタに転送アドレスと転送語数を与え，次のデータ・ブロックの存在を知らせるCNTビットをセットする．
	アレイチェイン・モード	メイン・メモリ上にアレイ・テーブルを作る．
	リンクアレイチェイン・モード	メイン・メモリ上にリンクアレイ・テーブルを作る．

●図4．50　DMACのアドレス・マップ

アドレス	D_{15}	D_{14}	D_{13}	D_{12}	D_{11}	D_{10}	D_9	D_8	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
E84000H	CSR0(チャンネル・ステータス・レジスタ)								CER0(チャンネル・エラー・レジスタ)							
	COC	BTC	NDT	ERR	ACT	DIT	PCT	PCS	0	0	0	エラー・コード				
E84004H	DCR0(デバイス・コントロール・レジスタ)								OCR0(オペレーション・コントロール・レジスタ)							
	XRM		DTYP		DPS	0	PCL		DIR	BTD	SIZE		CHAIN		REQG	
E84006H	SCR0(シーケンス・コントロール・レジスタ)								CCR0(チャンネル・コントロール・レジスタ)							
	0	0	0	0	MAC		DAC		STR	CNT	HLT	SAB	INT	0	0	0
E8400AH	MTC0(メモリ・トランスファ・カウンタ)															
E8400CH	MAR0(メモリ・アドレス・レジスタH)															
E8400EH	MAR0(メモリ・アドレス・レジスタL)															
E84014H	DAR0(デバイス・アドレス・レジスタH)															
E84016H	DAR0(デバイス・アドレス・レジスタL)															
E8401AH	BCT0(ベース・トランスファ・カウンタ)															
E8401CH	BAR0(ベース・アドレス・レジスタH)															
E8401EH	BAR0(ベース・アドレス・レジスタL)															
E84024H									NIV0(ノーマル・インタラプト・ベクトル)							
E84026H									EIV0(エラー・インタラプト・ベクトル)							
E84028H									MFC0(メモリ・ファンクション・コード・レジスタ)							
									0	0	0	0	0	FC2	FC1	FC0
E8402CH									CPR0(チャンネル・プライオリティ・レジスタ)							
									0	0	0	0	0	0	CP	
E84030H									DFC0(デバイス・ファンクション・コード・レジスタ)							
									0	0	0	0	0	FC2	FC1	FC0
E84038H									BFC0(ベース・ファンクション・コード・レジスタ)							
									0	0	0	0	0	FC2	FC1	FC0
E84040H ∫	チャンネル1　(E84000H～E84039と同じ内容)															
E84080H ∫	チャンネル2　(E84000H～E84039と同じ内容)															
E840C0H ∫	チャンネル3　(E84000H～E84039と同じ内容)															
E840FEH									GCR(ゼネラル・コントロール・レジスタ)							
									0	0	0	0	BT		BR	

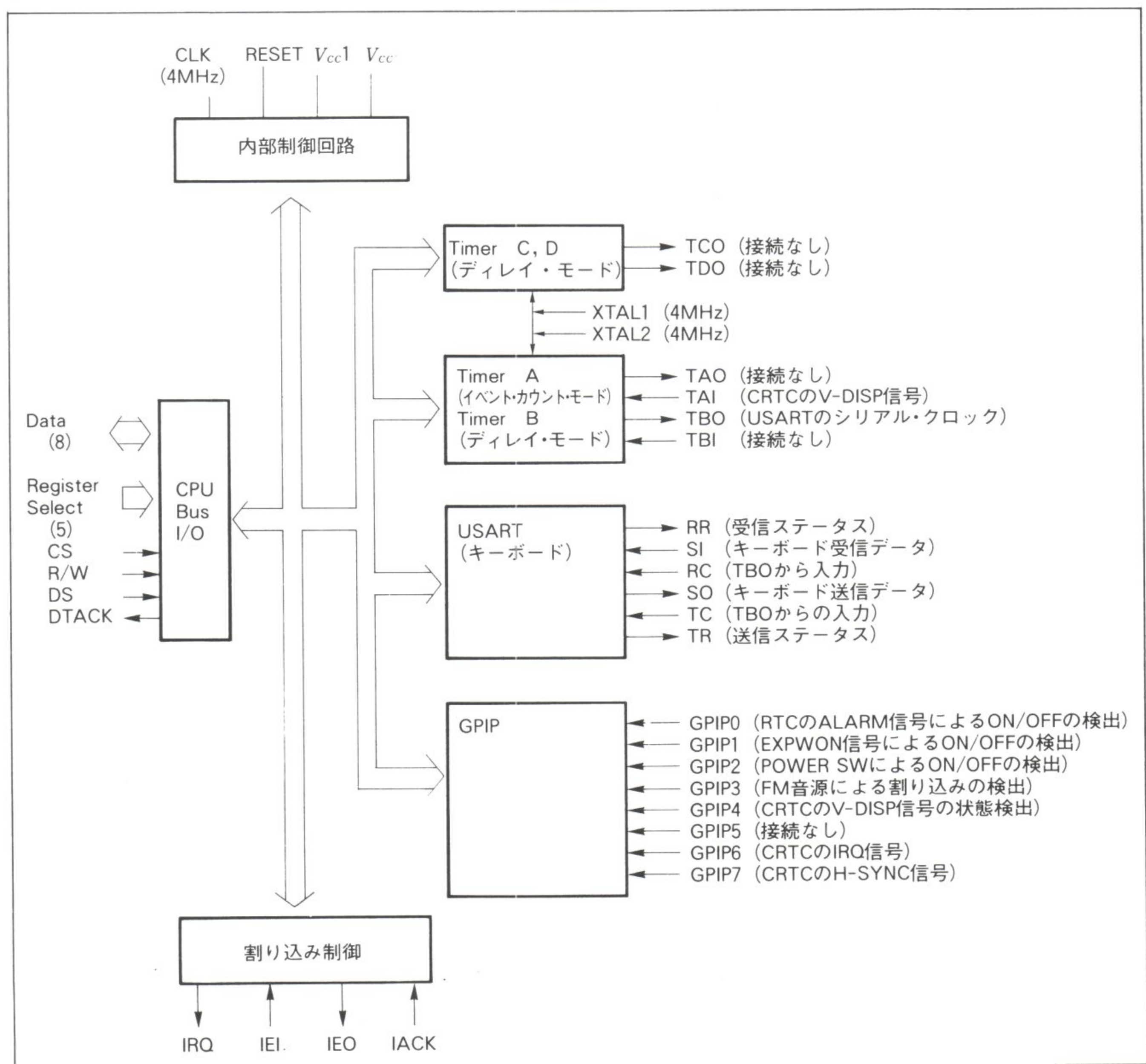
CERのみRead，その他はRead，Write可
E84000～E84039はチャンネル0

が不連続な場合は，ブロックごとに転送制御を継続する**コンティニュアス・モード**や，メモリ上に連続したブロック情報テーブルをもたせ，DMACがこれを読み取って自動的に継続する**アレイ・チェイン・モード**，分割しかつ関連づけされたブロック情報を読み取り自動継続する**リンク・アレイ・チェイン・モード**のいずれかを利用します(表4.15)．

X68000では，DMACは図4.50のアドレス帯に配置されています．チャンネル0～3は互いに独立しており，プライオリティによって優先づけされて動作します．したがって，制御情報も全チャンネル同じフォーマットになっています．個々のパラメータについては，DMACの説明書を参照してください．

4-13 MFPと接続環境

MFP(68901)とは「マルチ・ファンクション・ペリフェラル」の略で，モトローラにより，68000周辺デバイスとして開発されたICです．16ビット・パソコン用として設計されているため「図体」が大きく場所をとりますが，その代わり1つのパッケージの中に，4つの**タイマ**，全二重の非同期通信制御回路(USART)，16チャンネルの**割り込み制御回路**を収容することにより，全体としては装置の小型化に役立つように設計されています．

●図4．51　MFPブロック図

X68000のMFPの使われ方は図4.51のようになっており，USRTはキーボード用として割り付けられています(図4.52)．また，割り込みの制御は，図4.53の信号関係で集約され，68000に伝送されます．カウンタはTBOがUSARTのシリアル・クロックを生成し，TAIがCRTCのV-DISP信号がダウン・カウントして0になったら割り込みを発生する用途に使われています(図4.54)．

割り込みに関係する各チャンネルの利用のされ方の詳細は，表4.16に示すとおりです．この中には利用形態の性質上，必ずしも割り込みにつながらないものも含まれています．

MFPの接続環境で少し特殊なところでは，GPIP 0～2入力(GPIPは汎用入力ポート)を電源ONデバイスの識別に使っていることがあげられます．このことを利用してプログラムで特定化する方法を図4.55に示します．

●図4．52　MFP USART系ブロック図

TBO端子
RC端子
TC端子
USART（×16）
SI（キーボード受信データ）
SO（キーボード送信データ）
RR（受信ステータス）
TR（送信ステータス）
非同期通信
2,400 ボー
スタート1ビット，データ8ビット
パリティなし，ストップ1ビット

●図4．53　MFP 割り込み系ブロック図

68000MPU
IPL0
IPL1
IPL2
〈デコーダ〉
高
優先順位
低
INT7
INT6
INT5
INT4
INT3
INT2
INT1

〈68901MFP割り込み，読み出しチャンネル〉

優先順位		
高	15	GPIP7　(CRTCのH-SYNC信号)
	14	GPIP6　(CRTCのIRQ信号)
	13	TimerA（CRTCのV-DISP信号）
	12	Receive Buffer Full（キー・データの受信）
	11	Receive Error（キー・データの受信エラー）
	10	Transmit Buffer Empty（キー・データの送信）
	9	Transmit Error（キー・データの送信エラー）
	8	TimerB（USARTシリアル・クロック）
	7	GPIP5　(接続なし)
	6	GPIP4　(CRTCのV-DISP信号の状態検出)
	5	TimerC（8ビット汎用タイマ）
	4	TimerD（8ビット汎用タイマ）
	3	GPIP3　(FM音源による割り込みの検出)
	2	GPIP2　(POWER SWによるON/OFFの検出)
	1	GPIP1（拡張用I/OスロットからのEXPWON信号によるON/OFFの検出）
低	0	GPIP0　(RTCのALARM信号によるON/OFFの検出)

●図4．54　MFP タイマ系ブロック図

TADR
TAI（CRTCのV-DISP信号）
8ビット・カウンタ
1/2
TAO（接続なし）
イベント・カウント・モード
TACR（08H）
TBCR（01H）
TBDR（0DH）
(4MHz)
プリスケーラ(14)
8ビット・カウンタ
1/2
TBO
(USARTシリアル・クロック)
38400Hz
TCDR
TCDCR
(4MHz)
プリスケーラ
8ビット・カウンタ
1/2
TCO（接続なし）
TDDR
8ビット・カウンタ
1/2
TDO（接続なし）
ディレイ・モード

●表4．16　MFP各チャンネル詳細

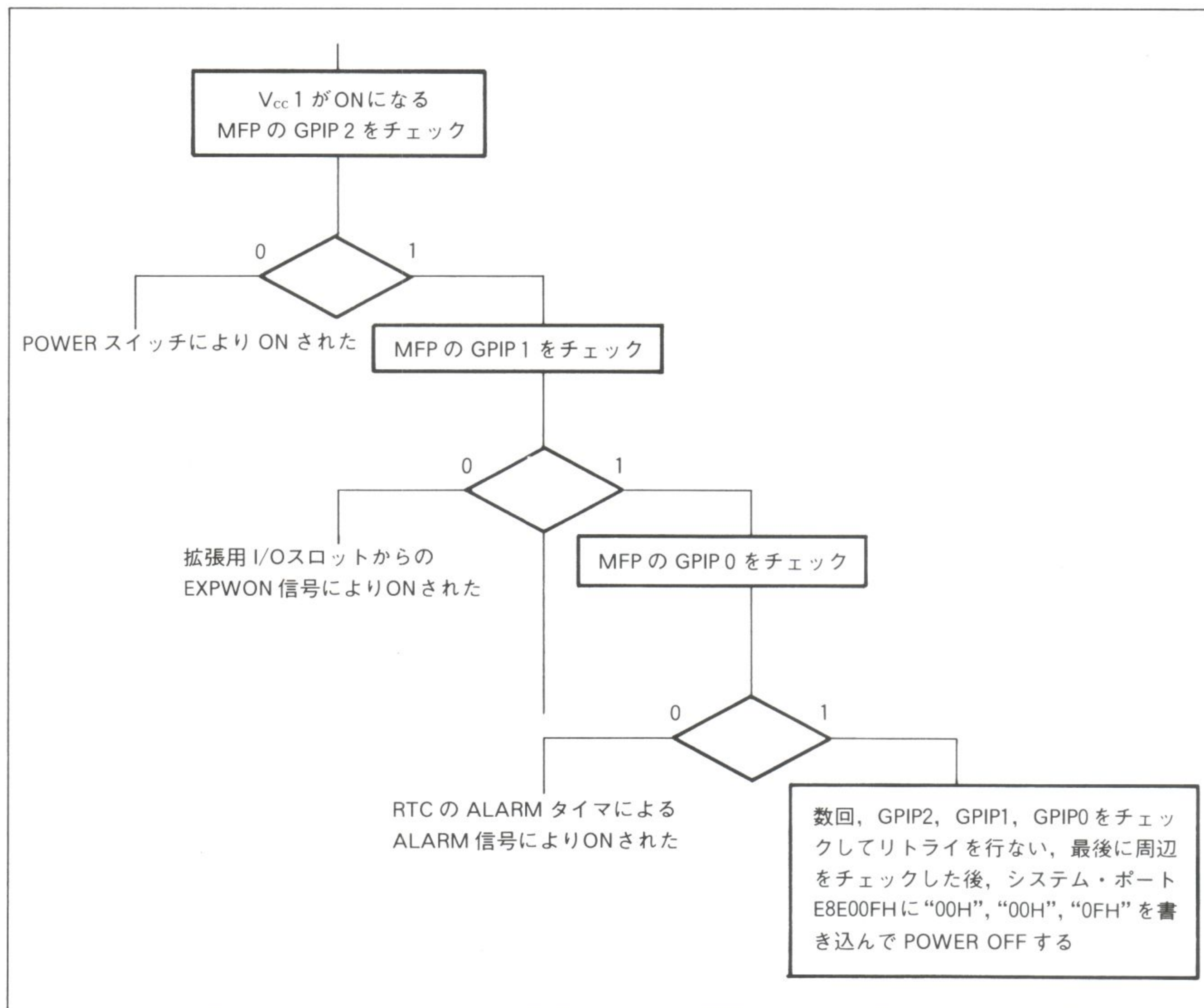
チャンネル No.	機　能　詳　細
GPIP7	CRTCのH-SYNC信号の立ち下がりにより割り込みを発生する．(アクティブエッジ・レジスタの値は，8をセット) H-SYNC
GPIP6	CRTCのR09(E80012H)で設定された割り込みラスター・アドレスで割り込みを起こしたい場合に，CRTCのIRQ信号の立ち下がりにより割り込みを発生する．(アクティブエッジ・レジスタの値は，8をセット) IRQ
TimerA	CRTCのV-DISP信号を入力としたタイマ(イベント・カウント・モード)において任意に設定されたカウント・パルス発生方法(入力信号の立ち下がりか，立ち上がりによりカウント・パルスを発生)とダウン・カウンタ値によりダウン・カウンタが00Hになったときに割り込みを発生する．
Receive Buffer Full	キーボードからデータを受信したとき(受信データが受信シフト・レジスタから受信バッファに転送され，受信ステータス・レジスタのD_7が1になったとき)に割り込みが発生する．
Receive Error	キーボードからデータを受信したときにエラー(オーバーラン・エラーまたは，パリティ・エラーまたは，同期検出，ブレーク検出)を起こした場合(受信ステータス・レジスタのD_7＝1または，D_5＝1または，D_3＝1のとき)に割り込みが発生する．
Transmit Buffer Empty	キーボードへデータを送信したとき(送信データが送信バッファから送信シフト・レジスタに転送され，送信ステータス・レジスタのD_7が1になったとき)に割り込みが発生する．
Transmit Error	キーボードへデータを送信したときにエラー(アンダーラン・エラーまたは，トランスミッタ終了)を起こした場合(送信ステータス・レジスタのD_6＝1または，D_4＝1のとき)に割り込みが発生する．
TimerB	内部クロック(4MHz)を入力としたタイマ(ディレイ・モード)において，MFPのUSART(キーボード)への入力クロックを作る(38,400Hz)．ただし，キーボードのシリアル・クロックに使用するので割り込みは不可．
GPIP5	予約ずみ
GPIP4	CRTCのV-DISP信号の状態を読み出す．1のときが，Hすなわち，垂直表示期間を示し，0のときが，Lすなわち，垂直帰線期間を示す(割り込み不可)．
TimerC	内部クロック(4MHz)を入力としたタイマ(ディレイ・モード)において，任意に設定されたプリスケーラとダウン・カウンタを使用して，ダウン・カウンタが00Hになったときに割り込みを発生する．
TimerD	内部クロック(4MHz)を入力としたタイマ(ディレイ・モード)において，任意に設定されたプリスケーラとダウン・カウンタを使用して，ダウン・カウンタが00Hになったときに割り込みを発生する．
GPIP3	FM音源のIRQ信号の立ち下がりにより割り込みを発生する．
GPIP2	POWERスイッチ(フロント電源スイッチ)によって，コンピュータの電源(Vcc1)がONされたかどうかを読み出す．通常あるいは，POWERスイッチがONの状態では，0すなわちLになっているが，POWERスイッチが押されてOFFになると，1すなわちHにかわる．ただし，このPOWERスイッチがOFFされたかどうかの検出方法については，通常この信号の立ち上がりによる割り込みを使用するのがよい．
GPIP1	拡張用I/OスロットからEXPWON信号を使ってコンピュータの電源(Vcc1)がONされたかどうかを読み出す．通常あるいは，EXPWON信号以外の方法によりコンピュータの電源がONされた場合は，1すなわちHになっているが，EXPWON信号を使用してコンピュータの電源がONされた場合は0すなわちLになる．つまり，MPUは，このGPIP1ポートを調べることにより，コンピュータの電源が拡張用I/OスロットのEXPWON信号を使用してONされたかどうかを知ることができる．
GPIP0	RTCのALARMタイマを使用してALARM信号を発生し，そのALARM信号によりコンピュータの電源がONされたかどうかを読み出す．通常は，1すなわちHになっているが，ALARM信号が発生したときには，1分間だけ0すなわちLになる．つまり，MPUは，コンピュータの電源がONになってから，1分間以内であれば，このGPIP0ポートを調べることにより，コンピュータの電源がALARM信号によってONされたかどうかを知ることができる．ただし，このALARM信号には，ALARMタイマによる信号のほかに，1Hz，または16Hzのクロック・パルスも使用（プログラマブル）でき，その信号の任意に設定された状態変化による割り込みを発生することもできる．

●図4．55　電源ONデバイス検出方法

V_{cc}1がONになる
MFPのGPIP 2をチェック

0：POWERスイッチによりONされた
1：MFPのGPIP 1をチェック

0：拡張用I/Oスロットからの
EXPWON信号によりONされた
1：MFPのGPIP 0をチェック

0：RTCのALARMタイマによる
ALARM信号によりONされた
1：数回，GPIP2，GPIP1，GPIP0をチェックしてリトライを行ない，最後に周辺をチェックした後，システム・ポートE8E00FHに"00H"，"00H"，"0FH"を書き込んでPOWER OFFする

MFPの詳細はマニュアルに譲るとして，ここではX68000での各レジスタのアドレス配置に限定してまとめたものを表4.17に示します．また，各レジスタの標準的なビット設定値を表4.18に示します．そして各レジスタのうち，ビットごとに意味をもつものについては，図4.56によりその詳細を説明します．

●表4．17　MFPレジスタ・アドレス・マップ

	レジスタ・アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備　考
GPIP コントロール	E88001H	GPIP7	GPIP6	GPIP5	GPIP4	GPIP3	GPIP2	GPIP1	GPIP0	GPIPデータ・レジスタ(Read only)
	E88003H	GPIP7	GPIP6	GPIP5	GPIP4	GPIP3	GPIP2	GPIP1	GPIP0	アクティブ・エッジ・レジスタ(AER)
	E88005H	GPIP7	GPIP6	GPIP5	GPIP4	GPIP3	GPIP2	GPIP1	GPIP0	データ・ディレクション・レジスタ(DDR)
割り込みコントロール	E88007H	GPIP7	GPIP6	TimerA	RCV Buffer Full	RCV Error	XMIT Buffer Empty	XMIT Error	TimerB	割り込みイネーブル・レジスタA (IERA)
	E88009H	GPIP5	GPIP4	TimerC	TimerD	GPIP3	GPIP2	GPIP1	GPIP0	割り込みイネーブル・レジスタB (IERB)
	E8800BH	GPIP7	GPIP6	TimerA	RCV Buffer Full	RCV Error	XMIT Buffer Empty	XMIT Error	TimerB	割り込みペンディング・レジスタA (IPRA)
	E8800DH	GPIP5	GPIP4	TimerC	TimerD	GPIP3	GPIP2	GPIP1	GPIP0	割り込みペンディング・レジスタB (IPRB)
	E8800FH	GPIP7	GPIP6	TimerA	RCV Buffer Full	RCV Error	XMIT Buffer Empty	XMIT Error	TimerB	割り込みインサービス・レジスタA (ISRA)
	E88011H	GPIP5	GPIP4	TimerC	TimerD	GPIP3	GPIP2	GPIP1	GPIP0	割り込みインサービス・レジスタB (ISRB)
	E88013H	GPIP7	GPIP6	TimerA	RCV Buffer Full	RCV Error	XMIT Buffer Empty	XMIT Error	TimerB	割り込みマスク・レジスタA (IMRA)
	E88015H	GPIP5	GPIP4	TimerC	TimerD	GPIP3	GPIP2	GPIP1	GPIP0	割り込みマスク・レジスタB (IMRB)
	E88017H	V_7	V_6	V_5	V_4	S				ベクタ・レジスタ
Timer コントロール	E88019H				Reset TAO	AC3	AC2	AC1	AC0	タイマAコントロール・レジスタ(TACR)
	E8801BH				Reset TBO	BC3	BC2	BC1	BC0	タイマBコントロール・レジスタ(TBCR)
	E8801DH		CC2	CC1	CC0		DC2	DC1	DC0	タイマC, コントロール・レジスタ(TCDCR)
	E8801FH	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	タイマAデータ・レジスタ(TADR)
	E88021H	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	タイマBデータ・レジスタ(TBDR)
	E88023H	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	タイマCデータ・レジスタ(TCDR)
	E88025H	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	タイマDデータ・レジスタ(TDDR)
USART コントロール	E88027H	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	同期キャラクタ・レジスタ(未使用)
	E88029H	CLK	WLI	WL0	ST1	ST0	PE	E/O	*	USARTコントロール・レジスタ(UCR)
	E8802BH	BF	CE	PE	FE	F/S or B	M/CIP	SS	RE	受信ステータス・レジスタ(RSR)
	E8802DH	BE	UE	AT	END	B	H	L	TE	送信ステータス・レジスタ(TSR)
	E8802FH	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	USARTデータ・レジスタ(UDR)

●表4．18　MFP各レジスタ標準設定値

レジスタ・アドレス	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	備　考
E88001H	*	*	*	*	*	*	*	*	GPIPデータ・レジスタ(Read only)
E88003H	0	0	*	P	*	1	*	*	アクティブエッジ・レジスタ(AER)
E88005H	0	0	0	0	0	0	0	0	データ・ディレクション・レジスタ(DDR)
E88007H	P	P	P	1	1	P	P	0	割り込みイネーブル・レジスタA (IERA)
E88009H	P	0	P	P	0	1	0	0	割り込みイネーブル・レジスタB (IERB)
E8800BH	*	*	*	*	*	*	*	*	割り込みペンディング・レジスタA (IPRA)
E8800DH	*	*	*	*	*	*	*	*	割り込みペンディング・レジスタB (IPRB)
E8801FH	*	*	*	*	*	*	*	*	割り込みインサービス・レジスタA (ISRA)
E88011H	*	*	*	*	*	*	*	*	割り込みインサービス・レジスタB (ISRB)
E88013H	P	P	P	1	1	P	P	0	割り込みマスク・レジスタA (IMRA)
E88015H	P	0	P	P	0	1	0	0	割り込みマスク・レジスタB (IMRB)
E88017H	P	P	P	P	P	*	*	*	ベクタ・レジスタ
E88019H	0	0	0	0	1	0	0	0	タイマAコントロール・レジスタ(TACR)
E8801BH	0	0	0	0	0	0	0	1	タイマBコントロール・レジスタ(TBCR)
E8801DH	0	P	P	P	0	P	P	P	タイマC, Dコントロール・レジスタ(TCDR)
E8801FH	P	P	P	P	P	P	P	P	タイマAデータ・レジスタ(TADR)
E88021H	0	0	0	0	1	1	0	1	タイマBデータ・レジスタ(TBDR)
E88023H	P	P	P	P	P	P	P	P	タイマCデータ・レジスタ(TCDR)
E88025H	P	P	P	P	P	P	P	P	タイマDデータ・レジスタ(TDDR)
E88027H	0	0	0	0	0	0	0	0	同期キャラクタ・レジスタ(未使用)
E88029H	1	0	0	0	1	0	0	*	USARTコントロール・レジスタ(UCR)
E8802BH	*	*	*	*	*	*	0	P	受信ステータス・レジスタ(RSR)
E8802DH	*	*	P	*	P	1	0	P	送信ステータス・レジスタ(TSR)
E8802FH	*	*	*	*	*	*	*	*	USARTデータ・レジスタ(UDR)

ただし，上表において，Pはプログラマブル，*は該当なし．

●図4．56　MFPレジスタ詳細①

(a)　GPIPコントロール

• GPIPデータ・レジスタ（E88001H）

MFPのGPIPポート（8ビット）は，ビットごとに，通常のポートに指定したり，割り込みのポートに指定したりできる．#1000では，GPIP0，GPIP1を読み出しポートに使用しているほか，タイマAにイベント・カウント・モードを使用しているので，GPIP4も読み出しポートに使用している．また，GPIP2，GPIP5，GPIP6，GPIP7（GPIP0）については割り込みポートとして使用している．いずれにしても，全ビット読み出しのみになっている。ただし，リセット時は全ビットが0になる．

D_7　D_6　D_5　D_4　D_3　D_2　D_1　D_0

D_0:
0 — RTCのALARM信号“L”（ALARMタイマによるコンピュータの電源ON）
1 — RTCのALARM信号“H”

D_1:
0 — 拡張用I/OスロットからのEXPWON信号“L”
（EXPWON信号によるコンピュータの電源ON）
1 — 拡張用I/OスロットからのEXPWON信号“H”

D_2:
0 — POWERスイッチ（フロント電源）ON“L”
1 — POWERスイッチ（フロント電源）OFF“H”

D_3:
0 — FM音源のIRQ信号“L”
1 — FM音源のIRQ信号“H”

D_4:
0 — CRTCのV-DISP信号“L”（垂直帰線期間）
1 — CRTCのV-DISP信号“H”（垂直表示期間）

D_6:
0 — CRTCのIRQ信号“L”
1 — CRTCのIRQ信号“H”

D_7:
0 — CRTCのH-SYNC信号“L”
1 — CRTCのH-SYNC信号“H”

• アクティブエッジ・レジスタ（E88003H）

D_7　D_6　D_5　D_4　D_3　D_2　D_1　D_0

D_0:
0 — RTCのALARM信号が1→0のとき割り込み発生
1 — RTCのALARM信号が0→1のとき割り込み発生
（通常は，読み出しポート）

D_1:
0 — 拡張用I/OスロットからのEXPWON信号が1→0のとき割り込み発生
1 — 拡張用I/OスロットからのEXPWON信号が0→1のとき割り込み発生
（通常は，読み出しポート）

D_2:
0 — POWERスイッチが1（OFF状態）→0（ON状態）のとき割り込み発生
1 — POWERスイッチが0（ON状態）→1（OFF状態）のとき割り込み発生（通常）

D_3:
FM音源のIRQ信号が1→0のとき割り込み発生

D_4:
0 — TAI（CRTCのV-DISP）信号が1→0のときタイマAカウント・パルス発生
1 — TAI（CRTCのV-DISP）信号が0→1のときタイマAカウント・パルス発生

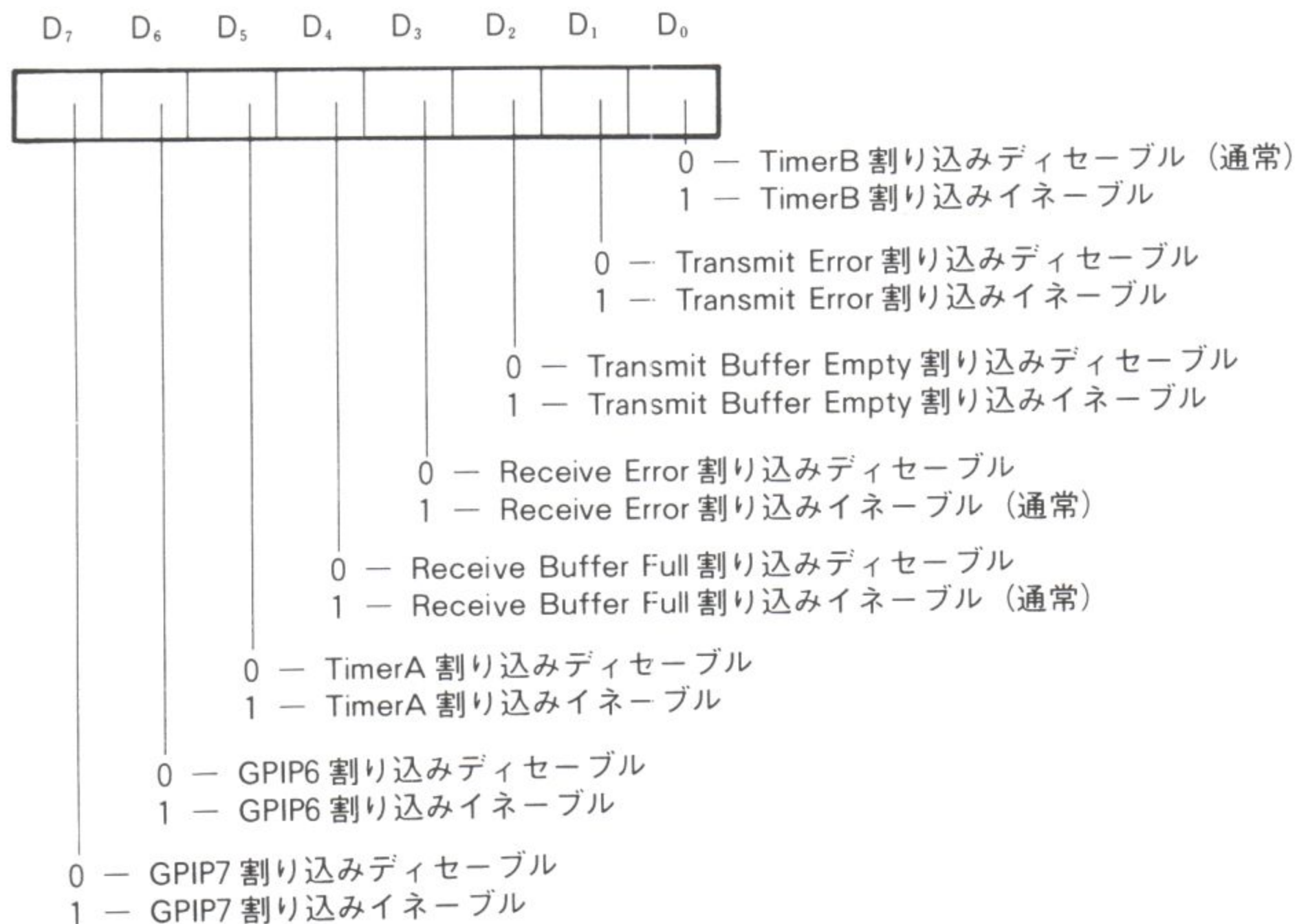
D_6:
0 — CRTCのIRQ信号が1→0のとき割り込み発生

D_7:
0 — CRTCのH-SYNC信号が1→0のとき割り込み発生

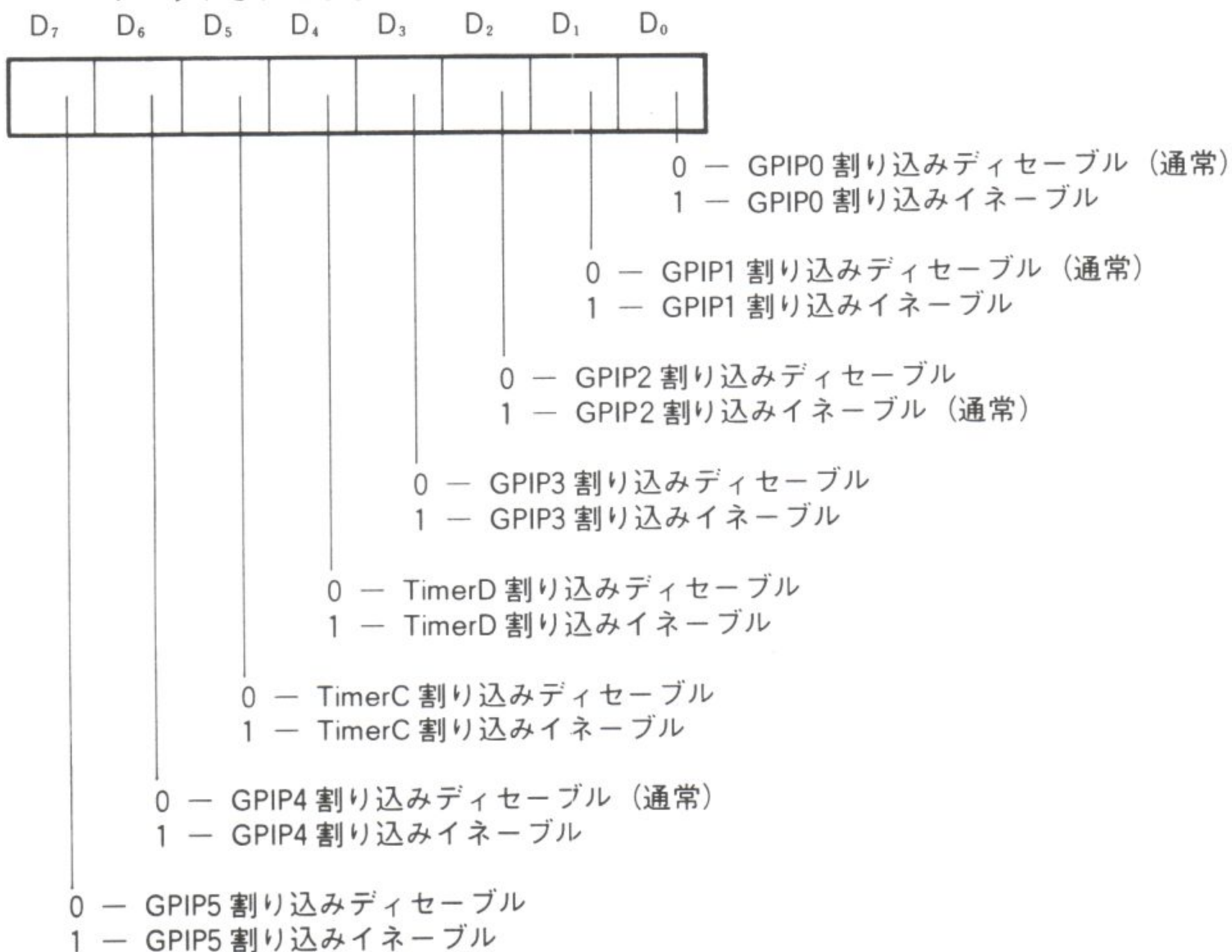
※ただし，リセット時は，全ビットが0

(b)　割り込みコントロール
・割り込みイネーブル・レジスタA（E88007H）
　a．該当ビットがセット(1)される場合
　　＊割り込みをイネーブルにするため，1 を書き込んだとき
　b．該当ビットがクリア(0)される場合
　　＊割り込みをディセーブルにするため，0 を書き込んだとき（IPRAの該当ビットも 0 になる）
　　＊リセットされたとき

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

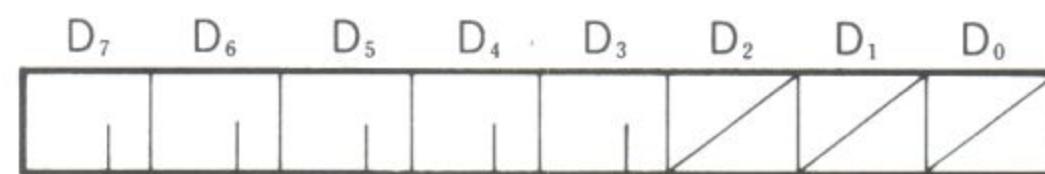
- D_0:
  0 ― TimerB 割り込みディセーブル（通常）
  1 ― TimerB 割り込みイネーブル
- D_1:
  0 ― Transmit Error 割り込みディセーブル
  1 ― Transmit Error 割り込みイネーブル
- D_2:
  0 ― Transmit Buffer Empty 割り込みディセーブル
  1 ― Transmit Buffer Empty 割り込みイネーブル
- D_3:
  0 ― Receive Error 割り込みディセーブル
  1 ― Receive Error 割り込みイネーブル（通常）
- D_4:
  0 ― Receive Buffer Full 割り込みディセーブル
  1 ― Receive Buffer Full 割り込みイネーブル（通常）
- D_5:
  0 ― TimerA 割り込みディセーブル
  1 ― TimerA 割り込みイネーブル
- D_6:
  0 ― GPIP6 割り込みディセーブル
  1 ― GPIP6 割り込みイネーブル
- D_7:
  0 ― GPIP7 割り込みディセーブル
  1 ― GPIP7 割り込みイネーブル

・割り込みイネーブル・レジスタB(E88009H)
　a．該当ビットがセット(1)される場合
　　＊割り込みをイネーブルにするため，1 を書き込んだとき
　b．該当ビットがクリア(0)される場合
　　＊割り込みをディセーブルにするため，0 を書き込んだとき(IPRAの該当ビットも 0 になる)
　　＊リセットされたとき

D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0

- D_0:
  0 ― GPIP0 割り込みディセーブル（通常）
  1 ― GPIP0 割り込みイネーブル
- D_1:
  0 ― GPIP1 割り込みディセーブル（通常）
  1 ― GPIP1 割り込みイネーブル
- D_2:
  0 ― GPIP2 割り込みディセーブル
  1 ― GPIP2 割り込みイネーブル（通常）
- D_3:
  0 ― GPIP3 割り込みディセーブル
  1 ― GPIP3 割り込みイネーブル
- D_4:
  0 ― TimerD 割り込みディセーブル
  1 ― TimerD 割り込みイネーブル
- D_5:
  0 ― TimerC 割り込みディセーブル
  1 ― TimerC 割り込みイネーブル
- D_6:
  0 ― GPIP4 割り込みディセーブル（通常）
  1 ― GPIP4 割り込みイネーブル
- D_7:
  0 ― GPIP5 割り込みディセーブル
  1 ― GPIP5 割り込みイネーブル

（続く）

●図5．56 MFPレジスタ詳細②

・割り込みベクタ・レジスタ（E88017H）

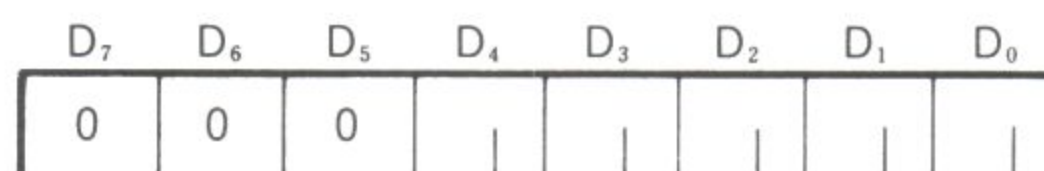
D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V7	V6	V5	V4		／	／	／

D_3: 0 ― 全チャンネル自動割り込み終了モード
1 ― 全チャンネル・ソフトウェア割り込み終了モード（ISRA，ISRB 有効）

V7 V6 V5 V4 ― 割り込みベクタ上位4ビット（下位4ビットは，各割り込みチャンネルNo.（0―16）にMFPが自動的に対応）

ただし，リセット時は，全ビットが0になる．

(c)　タイマ

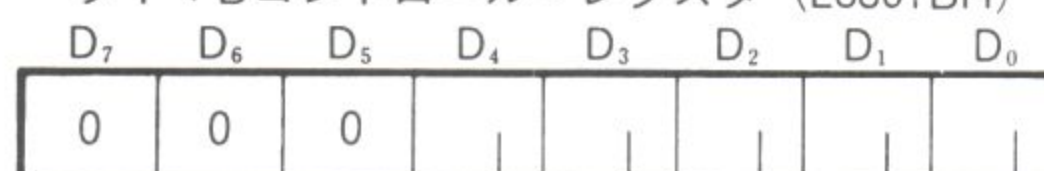
・タイマAコントロール・レジスタ（E88019H）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0					

D_3–D_0: 1　0　0　0 ― タイマAイベント・カウント・モード

D_4: 0 ― TAO信号リセットしない
1 ― TAO信号リセットする

ただし，リセット時は，全ビットが0になる．

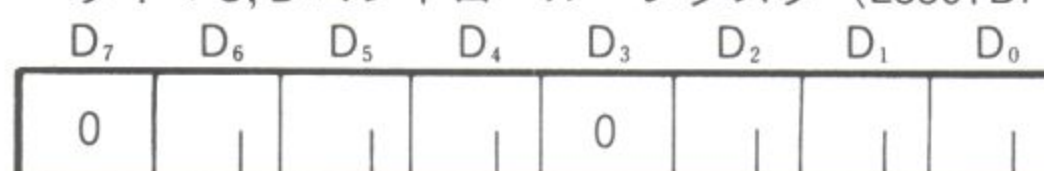
・タイマBコントロール・レジスタ（E8801BH）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0					

D_3–D_0: 0　0　0　1 ― タイマBディレイ・モード（/4プリスケーラ）

D_4: 0 ― TBO 信号リセットしない
1 ― TBO 信号リセットする

ただし，リセット時，全ビットが0になる．

・タイマC,D コントロール・レジスタ（E8801DH）

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0				0			

D_2	D_1	D_0	
0	0	0	― タイマDストップ（カウント禁止）
0	0	1	― タイマDディレイ・モード（/4プリスケーラ）
0	1	0	― タイマDディレイ・モード（/10プリスケーラ）
0	1	1	― タイマDディレイ・モード（/16プリスケーラ）
1	0	0	― タイマDディレイ・モード（/50プリスケーラ）
1	0	1	― タイマDディレイ・モード（/64プリスケーラ）
1	1	0	― タイマDディレイ・モード（/100プリスケーラ）
1	1	1	タイマDディレイ・モード（/200プリスケーラ）

D_6	D_5	D_4	
0	0	0	― タイマCストップ（カウント禁止）
0	0	1	― タイマCディレイ・モード（/4プリスケーラ）
0	1	0	― タイマCディレイ・モード（/10プリスケーラ）
0	1	1	― タイマCディレイ・モード（/16プリスケーラ）
1	0	0	― タイマCディレイ・モード（/50プリスケーラ）
1	0	1	― タイマCディレイ・モード（/64プリスケーラ）
1	1	0	― タイマCディレイ・モード（/100プリスケーラ）
1	1	1	― タイマCディレイ・モード（/200プリスケーラ）

ただし，リセット時は，全ビットが0になる．

(1) MFPのRC，TC端子には，16倍のクロックを入力
(2) ボーレートは，2,400ボー
(3) スタート1ビット，データ8ビット，パリティなし，ストップ1ビット

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
1	0	0	0	1	0	0	*

(*には，WRITE時は無効，READ時は0が入る)

ただし，リセット時は，全ビットが0になる．

• 受信ステータス・レジスタ (E8802BH)

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
						0	

D_0:
0 ― “0”を書き込むとすべてのフラグがクリアされレシーバはディセーブルになる．
1 ― “1”を書き込むとレシーバがイネーブルになる．

D_2:
0 ― ストップ・ビットを受信
1 ― スタート・ビットを受信

D_3:
0 ― ブレーク・キャラクタ以外のデータを受信し，少なくとも1回RSRを読み出した場合
1 ― 受信シフト・レジスタから受信バッファに転送されたデータがブレーク・キャラクタ（ストップ・ビットをもたない8ビット全部が0のキャラクタ）の場合

D_4:
0 ― 正常な受信データ
1 ― 受信データ・フレーム・エラー（ストップ・ビットが検出できない場合）

D_5:
0 ― 正常な受信データ
1 ― 受信データ・パリティ・エラー（受信シフト・レジスタから受信バッファに転送されたデータが，パリティ・エラーを起こしていた場合 ）

D_6:
0 ― RSRを読み出した場合
1 ― 受信データ・オーバーラン・エラー（受信データが，受信シフト・レジスタから受信バッファに転送されるときに，受信バッファがフル状態の場合）
なお，UDRが読み出されてバッファフルの状態が解除された後，このビットはセットされる．

D_7:
0 ― 受信バッファ（UDR）が読み出された場合（Read only）
1 ― 受信データが受信シフト・レジスタから受信バッファに転送された場合（Read only）

なお，D_7=1(バッファフル）のときには，Receive Buffer Full（MFP 12レベルの割り込み）が起こり，D_6＝1（オーバーラン・エラー）あるいはD_5＝1（パリティ・エラー）あるいは，D_3＝1（ブレーク検出）のときには，Receive Error（MFP 11レベルの割り込み）が起こる。また，Receive Frrorの割り込みが，ディセーブルのときは，たとえエラーが起こっていたとしても，Receive Buffer Fullの割り込みが起こる．

ただし，リセット時は，全ビットが0になる．

・送信ステータス・レジスタ(E8802DH)

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
					1	0	

D_0:
0 ―“0”を書き込むとD_6＝0，D_4＝1になりトランスミッタはディセーブルになる．
1 ―“1”を書き込むとトランスミッタがイネーブルになる．
なお，データの送信が始まる前に1ビットが送信される．

D_3:
0 ―通常の送信モード
1 ―ブレーク･キャラクタ(ストップ･ビットをもたない8ビット全部が0のキャラクタ)送出
なお，このときは，D_7はセットされないので注意が必要

D_4:
0 ―D_0＝1（トランスミッタ・イネーブル)に設定された場合
1 ―D_0＝0（トランスミッタ・ディセーブル)に設定された場合(送信中は送信終了後セット)

D_5:
0 ―D_0＝0（トランスミッタ・ディセーブル)になった場合
1 ― 1を書き込むとトランスミッタ・ディセーブルとなり送信中の最終キャラクタが送信し終わった後，自動的にレシーバがイネーブル(RSRのD_0＝1）になる．

D_6:
0 ―TSRを読み出すか，トランスミッタをディセーブル(D_0＝0)にした場合
1 ―送信データ・アンダーラン・エラー(UDRから送信バッファに送信データが転送される前に送信シフト・レジスタからすでに送信データが送信されていた場合)

D_7:
0 ―UDRに書き込みを行なって送信バッファに送信データが転送された場合
1 ―送信バッファから送信シフト・レジスタに送信データが転送された場合

なお，D_7＝1（バッファ・エンプティ)のときには，Transmit Buffer Empty(MFP10レベルの割り込み)が起こり，D_6＝1(アンダーラン・エラー)あるいは，D_4＝1（トランスミッタ終了)のときには，Transmit Error(MFP 9 レベルの割り込み)が起こる．

ただし，リセット時は，全ビットが0になる．

4-14 SCCについて

SCC(Z8530)はシリアル通信用のICで，MFPのUSARTのような機能を2系統と，**ボーレート・ゼネレータ**を内蔵しています．68000ではこれによって**RS-232C**ポートと**マウス**に対応しており，MFPの拡張のために使われているともいえます．

ハードウェア・ブロックの関連は，図4.57のとおりで，外部ではバッファを経由してコネクタに接続されているだけというシンプルな構成になっています．

レジスタのアドレス・マップは図4.58に示すように，2系統のポートに対応するコマンドとデータ・ポートからなっています．

●図4．57　SCCブロック図

●図4．58　SCCレジスタ・アドレス・マップ

Ch No.	レジスタ・アドレス	D_7 D_6 D_5 D_4 D_3 D_2 D_1 D_0	備　考
B	E98001H	←→	コマンド・ポート
	E98003H	←→	データ・ポート
A	E98005H	←→	コマンド・ポート
	E98007H	←→	データ・ポート

あとの詳しいことは，SCCの素子そのものに依存する説明となってしまうので，本書ではマニュアルに譲りたいと思います．ただし，X68000におけるボーレート設定に関しては，表4.19の時定数データを書き込むこととなるので，この点だけ付け加えておきます．

●表4．19　各ボーレートに対するボーレート・ジェネレータの時定数

ボーレート	5MHzの場合の時定数	備　考
9,600	14(000EH)	時定数 $= \dfrac{5\times10^6}{2\times(ボーレート\times16)} - 2$
4,800	31(001FH)	
2,400	63(003FH)	
1,200	128(0080H)	
600	258(0102H)	
300	519(0207H)	ただし，入力クロック(PCLK)を5MHzとし，データ速度の16倍のクロックを用いた場合とする．
150	1,040(0410H)	
75	2,081(0821H)	

第2部

Human68Kの操作法

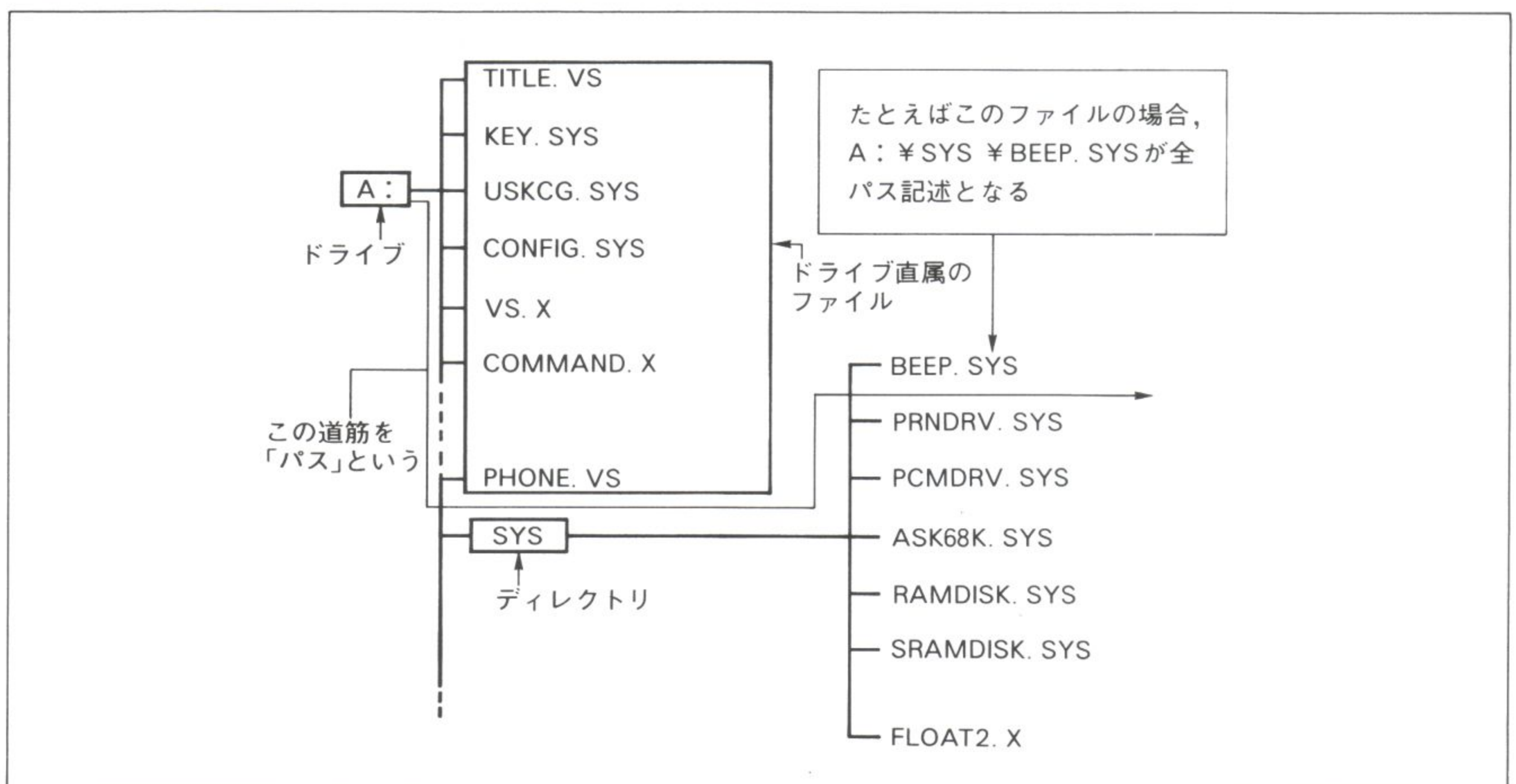
第1章

Human68K の基本的な機能とファイル管理

1-1 階層構造のディレクトリと拡張子

Human68K のファイル管理では，デバイス名を起点にして**ディレクトリ**，**ファイル名**とたどって目的のファイルを探し当てるようになっています．

ディレクトリとは，ファイル名を登録したファイルのようなもので，ディレクトリの下に別なディレク

●図1．1　階層構造のディレクトリによるファイル管理

"Human68K"は，いわば"MS-DOS"の X-68000版です．MS-DOS は，"UNIX"から派生したモニタ・システムで，階層構造のファイル管理を特徴としています．

本章では，Human68K の屋台骨ともいえるファイル管理の概念，コマンドなどについて詳述します．MS-DOS に不慣れな読者は，とりあえず本章だけマスターしておけば，かなりの程度の操作ができるようになります．

トリを置くことができます．その結果，ディレクトリは**階層構造**となり，ファイルはそれらの末端に位置することになります(図1.1)．

ディレクトリは仕事の単位とかファイルの目的別に設定され，同名ファイルでも異なるディレクトリの下では，まったく独立したものとして扱われます．言わば，ファイル名は1つのディレクトリ内でさえ衝突しなければ，同じものをいくつ定義してもかまいません．

MS-DOS 系のファイル管理の特徴は，ファイル名に続いて3字以内の**拡張子**を付けるところにあります．拡張子はファイルの種類を宣言するもので表1.1のものがシステムで規定されています．これによって，同

●表1．1 Human68K で使用している拡張子

拡張子	ファイルの種類
.SYS	システム・ファイル
.BAT	バッチ・ファイル
.BAS	BASIC ソース・ファイル
.C	C ソース・ファイル
.S	アセンブラ・ソース・ファイル
.O	オブジェクト・ファイル(アセンブラ出力)
.A	アーカイブ・ファイル
.X	ソフト・リロケータブル実行ファイル
.R	リロケータブル実行ファイル
.Z	絶対アドレス形式実行ファイル
.BAK	バックアップ・ファイル(エディタの更新前など)
.FNC	BASIC 外部関数ファイル
.DOC	ドキュメント・ファイル(Type して見るだけ)
.h	C の INCLUDE ファイル
.VS	画像イメージ・データ

●表1．2　予約ずみファイル名

ファイル名	用　途
AUX	補助入出力装置との入出力を指定するときに使用
CON	キーボードからの入力，またはスクリーンへの出力を指定するときに使用
PRN	プリンタへの出力を指定するときに使用(漢字 IN/漢字 OUT コードを出力する)
LPT	プリンタへの出力を指定するときに使用 (漢字 IN/漢字 OUT コードを出力しない．ビット・イメージ出力などに利用)
PCM	ADPCM デバイスとの入出力を指定するときに使用
CLOOK	Human68K 内部で使用されるファイル名．ユーザは使用不可
NUL	コマンドが入出力のファイル名を必要としているが，とくにファイルを作成しないときに使用

じプログラム(たとえば“abc”)のファイルでも，アセンブラ・ソースでは“abc.s”，アセンブル結果のオブジェクト・ファイルは“abc.o”，実行ファイルでは“abc.x”のように独立したものとなります．

拡張子は表以外でもアルファベット1字以外ならば任意に決めることができますが，あまり勝手に付けるとコマンドなどの指定で省略できるケースでも省略できなくなります．また，ファイル名についても，表1.2のものは予約ずみのため，ユーザが命名するときに使うことはできません．

1-2　カレント・ドライブとカレント・ディレクトリ

❖関連コマンド　:, chdir

Human68K では，コマンドを入力する際，現在対象にしているドライブを明確にするために，プロンプト(入力促進文字列)としてドライブ名を表示します．たとえば，

A>

はドライブ A を表わし，ドライブ名を省略したパス記述で暗黙にドライブ A が使われることを示しています．このようにドライブ名省略値はシステムで記憶されており，変更することも可能です．このドライブ名のことを**カレント・ドライブ**といいます．カレント・ドライブの変更は，

A>**B：**　(B に変更する)

のようにして行なえます．

ディレクトリについても同様なことがいえます．

階層構造のファイル管理では，ファイル名が衝突する心配がない代わりに，ファイル・パス記述が長くなります．普通，ディレクトリは仕事の単位などで設けられているので，一般にプログラムに関連するファイルは，同じディレクトリ下に集中する傾向にあります．このようなときは，暗黙にディレクトリ名を与える**カレント・ディレクトリ**を使えば，末端のファイル名だけの指定ですむので大変便利です．

その際，ファイル名の前には“¥”を付けないように注意しなければなりません．“¥”はディレクトリ名，ファイル名を連記するときの**セパレータ**(区切り)であると同時に，最初の“¥”はカレント・ディレクトリを参照しない全パス記述であることを示します．

カレント・ディレクトリは，chdir(短縮形：cd)コマンドによって変更できます．このとき，ディレクトリ名の前に“¥”を付けると，カレント・ドライブ名に続く全パス記述とみなされ，以前のカレント・ディレクトリ値は消去されます．また“¥”なしでは，カレント・ディレクトリの値に続けた一段下のディレクトリ名を指定する意味をもちます．なお，ディレクトリ名に代えて“..”を指定すると，一段上のディレクトリに戻ることができます．

1-3 コマンドの実行と外部コマンド・ファイルの扱い

❖関連コマンド　path

コマンドには，メーカー提供プログラムでCOMMAND. Xに収容されている**内部コマンド**と，ユーザなどが作成した実行形式のプログラムを働かせる**外部コマンド**の2種類があります(図1.2).

内部コマンドはいつでも実行できるよう配慮されています．外部コマンドは基本的に，カレント・ディレクトリが一致したとき，ファイル名だけ(拡張子は不要)の指定で実行できます．

これは，内部コマンドの働きが汎用的なのに対し，外部コマンドは対応するディレクトリの使用目的に依存するところが大きいためです．言い換えると，外部コマンドは入力されたコマンドの実行にあたって，最初にそのファイルを探します．このときファイル数があまりたくさんあると応答に時間がかかるため，多くなりがちな外部コマンドの対象を絞り込むことによって合理化を図っているのです．

しかし，外部コマンドでも汎用的なものもあり，またディレクトリの個々の目的とは関係なく使いたいプログラムも存在します．こういったケースを救済するために，

```
path [；]  [[<ドライブ>：] <パス名>]  {；[<ドライブ>：] <パス名>}
```

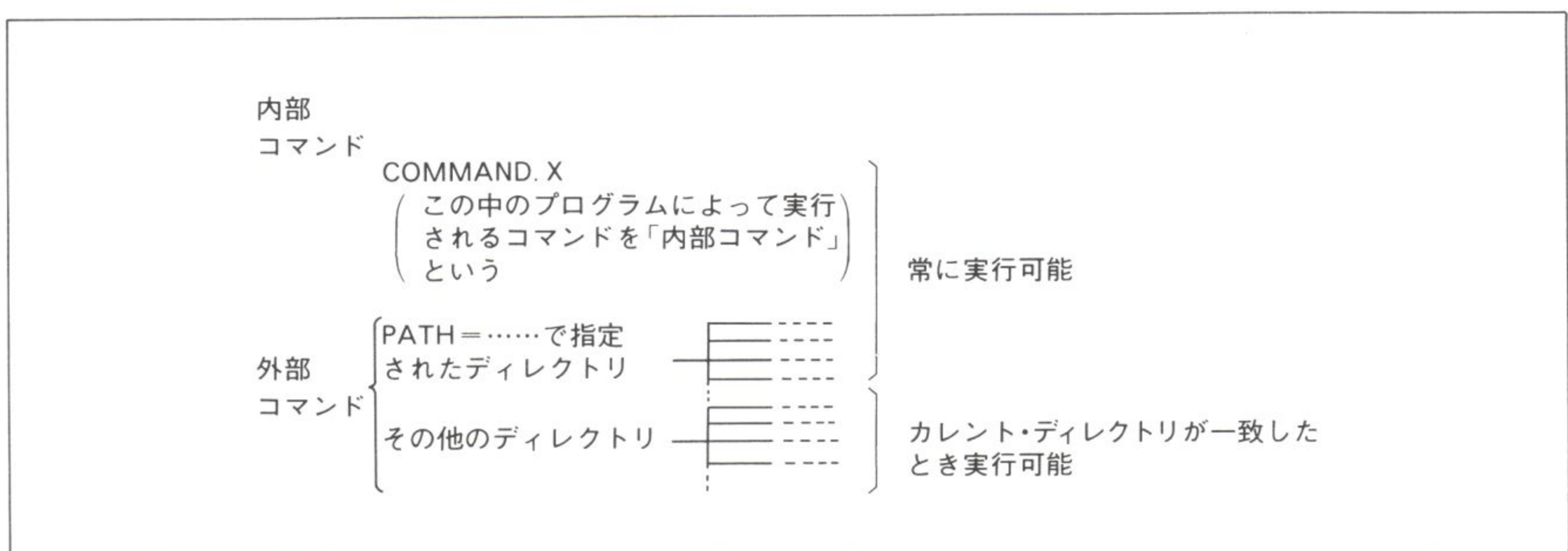
コマンドで指定されたディレクトリも検索範囲に含めるようになっています．しかし，あまりたくさんのディレクトリを指定すると，応答を遅くすることにつながりますから，できるだけユーザが作成する分は1つに絞るのが適当です．たとえば筆者の場合，“MyProgs”というディレクトリを用意し，汎用的なものはそこに入れるようにしています．

なお，現在実行可能なディレクトリにないプログラムでも，パス記述(“¥” で始まり，ディレクトリ名を付けたもので，必要があればドライブ名も前に置く)も含めたファイル名の指定をすることによって実行することはできます．

pathコマンドを実行すると，以前のpath指定は無効となります．また，“；”だけの記述は，何も指定しない(クリアする)ことを意味します．

●図1．2　ディレクトリ名を省略してコマンドが実行できる範囲

内部コマンド
COMMAND. X
(この中のプログラムによって実行されるコマンドを「内部コマンド」という)
常に実行可能

外部コマンド
PATH＝……で指定されたディレクトリ
その他のディレクトリ
カレント・ディレクトリが一致したとき実行可能

1-4 標準入出力とリダイレクト

たとえば，BASICのPRINT命令はスクリーンに，LPRINT命令はプリンタへというように，出力するデバイスが特定されています．このようなシステムでは，都合によってどちらでも好きなデバイスに出力できるようにするため，あらかじめプログラムに両方の機能を組み込み，かついずれかを選択する方法を

用意しておかなければなりません．

この点で**標準入力**，**標準出力**はいわば，「仮想デバイス」です．とくに指定がなければ，標準入力は**キーボード**に，標準出力は**スクリーン**に接続されています．そして，接続を換えるには，コマンド行に**リダイレクト記号**を使うことによって簡単にできるのが特徴です．

リダイレクト記号は，標準入力には“<”，標準出力には“>”を使います．標準出力としてディスクを使い，ファイルに続けて書き加える(アペンド)ときは“>>”の指定もできます．これらの記号の右側に接続先のパスを書きます．たとえば，標準出力をディスクに接続するときはファイル名までを，プリンタに接続するときはprnなどのデバイス名を記述します．

このような機能が使えるのは，入力，出力とも1つずつに限定されており，従来どおりデバイスの種類を特定した入出力方法が残されているのも事実です．しかし，可能な限り，プログラムでは標準入出力を使うように設計しておくことが後日の運用を柔軟にするコツです．

1-5 マルチ処理とパイプ

Human68Kでは，1つのコマンド行に複数のコマンドをまとめて記述できます．そして，個々のコマンド記述の間は“|”によって区切られます．

マルチ処理は“|”を2個並べて書くパターンで，並記されたコマンドは左側から順次実行されます．すなわち，

〈コマンド1〉 || 〈コマンド2〉 || 〈コマンド3〉

は，コマンド1，2，3をそれぞれ単独で順番に入力したのと同じ結果になります．1つだけ異なる点は，マルチ処理の場合，全部のコマンドが終了するまでコマンド・プロセッサ(COMMAND.X)が連続して動作することです．このことはたとえば，エディタから抜け出してコマンドを実行するなど，プログラム内からCOMMAND.Xを利用して複数コマンドを実行するとき，1回の手続きですむことを意味します．

パイプは“|”を1個だけ書くもので，

〈コマンド1〉 | 〈コマンド2〉 | 〈コマンド3〉

は，コマンド1の標準出力をコマンド2の標準入力に，コマンド2の標準出力をコマンド3の標準入力に接続することを表わします．この形態では，並記されたコマンドがあたかも同時に実行されているように見えます．実際はコマンド2はコマンド1からデータが転送されてくるまで待たされ，コマンド3でもコマンド2からのデータを受けるまで待ち合わせが入ります．この形態では，コマンド1だけの実行が終了しても全体は終了しません．すべてのコマンドが終了したときが，全体の終了となります．

マルチ処理では，単独でコマンドを起動する場合と同様に，リダイレクト記号が使えます．パイプの場合はコマンド間でリダイレクトが行なわれていることになりますが，矛盾が起きない範囲でほかにリダイレクトすることは差し支えありません．

1-6 ファイルの新設，削除，名称変更，リスト出力

❖関連コマンド　　**del, type, rename**

ファイルは，プログラムの出力でクリエイト(新設)モードにおいて作成されます．

そして，不要なファイルを削除するには，

ワイルド・カード(＊，?)について

コマンド記述上で，ファイル名などに“＊”を使用し，該当位置の文字列値を問わないで実行させる方法があります．たとえば，

del abc.＊

は，拡張子がどんな値であれ，上位のファイル名が“abc”なら消去してしまいます．この場合“＊”に対応する部分の字数は問いませんが，必要ならば“？”を使って字数を指定(“？”の数が字数となる)することもできます．字数指定のケースでは，指定された字数以下の文字列のみ該当するものとみなされ，オーバーしたものは無視されます．

このような機能を，「**ワイルド・カード**」といいます．

del 〈ファイル名〉

コマンドを使います．その場合は，dir コマンドでファイル名を確認するとか，必要によっては，

type 〈ファイル名〉

でファイルの内容を表示して点検し，消去しても困らないものであるかどうか充分調べておくにこしたことはありません．ただし，type コマンドは実行形式のプログラム・ファイルを対象にすることができないので，別の方法(たとえば dump コマンドなど)で調べます．調べる以前に明らかに不要なものであれば，dir コマンドでファイル名を調べる程度でも充分です．

ファイルの名称変更は，次のコマンドによってできます．

rename 〈旧ファイル名〉　〈新ファイル名〉　(rename：短縮形 ren)

これらのコマンドでは，カレント・ディレクトリを利用する(そのファイルを含む位置に置く)ことによって，ファイル名指定を簡略化できます．これは ren コマンドでも同様ですが，ren コマンドの場合，旧ファイル名を全パス・リストで指定したときでも，新ファイル名は“¥”から書き始める必要はありません．なぜなら，ファイル名の変更は，あくまで同じディレクトリ内において行なうのであって，他のディレクトリは関係ないからです．同じディレクトリならば，新旧ファイル名で重複して指定する必要はありません．

Human68K では，エディタその他で変更前の旧ファイルや，中間ファイルなどを残すため，あっという間に不要ファイルがどんどん増えていきます．このため，適当な時期を見て del コマンドで消さないと，すぐにディスクがパンクしてしまいます．その際，くれぐれもファイル名(拡張子を含め)を間違えないよう注意してください．

1-7 ディレクトリの新設，削除，リスト出力

❖関連コマンド　　mkdir, rmdir, dir

任意のディレクトリを**新設**するには，その親となるディレクトリをカレント・ディレクトリとし，

mkdir 〈子ディレクトリ名〉　(mkdir：md 短縮形 md)

コマンドを使うのが普通です．しかし，現在カレント・ディレクトリが別なところにあって移動させるの

が面倒なときは，全パス・リスト("¥"から始まる)を指定して強行することもできます．
　不要になったディレクトリを削除するには，同様に親ディレクトリの位置から，

　　rmdir 〈子ディレクトリ名〉　(rmdir：短縮形 **rd**)

コマンドを投入します．そのディレクトリにファイルや子ディレクトリが存在しないように，前もって **del** コマンド(ファイル消去)などで消しておく必要があります．
　また，全パス・リストを使用する場合には，制限事項があります．すなわち指定パス・リストがカレント・ディレクトリを含んでいるときは，rmdir が使えません．なぜなら，その結果残っているディレクトリとカレント・ディレクトリとが一致しなくなるからです．この意味で，ディレクトリ指定を"．．"(1つ上のディレクトリ)にもできません．
　ディレクトリの新設や消去に際しては，現在のディレクトリの内容を調べる(**表示**する)ことが必要です．また，このことは，ファイルを新設する際(または必要によっては，削除する場合も)にもいえることで，このようなときには，

　　dir [〈ディレクトリ名〉]

コマンドを使います．カレント・ディレクトリを利用する限りは，ディレクトリ名を省略することもでき，子ディレクトリを指定するときは"¥"なしで行ないます．
　dir コマンドを使用すると，ドライブでの使用領域，残り領域の大きさがKB(キロ・バイト)単位で表示されます．
　なお，ディレクトリ名を変更するときは，ファイル名と同様に **rename** コマンドが適用できます．

1-8 メディアの新設とファイルの保守

❖関連コマンド　**format, diskcopy, copy, fc, sys, attrib, vol, chkdsk**

　システム・ファイルなどのバックアップをとることは常識ですが，プログラムの開発環境が整ってくるにつれて，新しいファイルがどんどん増え，使用中のフロッピーに収容しきれなくなることがしばしばあります．このようなときは，別なメディアを調達して，その中に最小限必要なファイルを「引っ越し」させ，新規にスペースを確保しなければなりません．
　この場合調達したメディアがまったくの新品ならば，

　　format [〈ドライブ〉：] [／S] [／H] [／C] [／V]

によって，トラック，セクタの「枠組み」をするところから始めなければなりません．新品以外でも，他機や Human68K 以外のシステムで使用したメディアを使用するときは，フォーマットが必要です．
　フロッピーでは一般に，スイッチは省略してかまいませんが，必要があるときは次のように指定します．

　　／S……システム領域のコピーも行なう
　　／H……ハード・ディスクのとき
　　／C……フォーマットずみのメディアの初期化
　　／V……ボリューム・ラベルを付けるとき(省略時"Human68K")

　ここで，／Cスイッチは，すでに，フォーマットされているメディアの全ファイルを消去するためのものです．消去といってもルート・ディレクトリとFATの内容をクリアするだけなので，処理はあっという間に終わります．

単なるバックアップならば，このあと，

　　diskcopy ［〈コピー元ドライブ〉：　〈コピー先ドライブ〉：］［／V］

で全体のコピーをとれば完了しますが，新しいスペースを求めて「転居」するときは，ファイルの取捨選択を行なうため，少し時間がかかります．

1つの方法は，diskcopy で全部コピーしておいて，不要なファイルやディレクトリを del,rmdir(rd)によって消去することです．このとき不要ディレクトリのファイルは，そこにカレント・ディレクトリを置いて，

　　del ＊．＊（ワイルド・カード指定）

コマンドを投入すれば一発で消せます．

もう1つの方法は，format 後，個別にファイルを転送するもので，

　　copy 〈転送元記述〉〈転送先記述〉［／V］

コマンドによります．転送元，転送先は，ドライブ，パス(ディレクトリ)，ファイルの各レベルが指定でき，転送元がドライブならば，全ディレクトリ，全ファイルが対象となります．このとき diskcopy と異なるのは，個別のファイルが順次転送されるため，転送元に消去した跡があったり，ディスク・アドレスの「飛び」があっても転送先には連続的に埋められ，スペースが整理されることです．その代わり，スピードは遅くなります．

ディレクトリ・レベルでの転送は，そのディレクトリに属するファイルが全部まとめてコピーされます．ファイル・レベルでは指定されたファイルのみで，ワイルド・カードを使えば該当するものすべてが対象となります．

なお，diskcopy の／V スイッチは，コピーでなく，内容比較を行なわせるときに使います．これに対し copy の／V スイッチは，コピーしながらベリファイすることを指定するものです．

もし単純にファイル内容の比較だけを行ないたければ，

　　fc 〈ファイル名1〉〈ファイル名2〉［／A］［／B］［／W］［／C］［／n］

コマンド(ファイル・コンベア)を使います．比較は通常アスキー文字(／A)として行単位で行なわれますが，／W スイッチを指定すると，連続したブランクを1字とみなして比較します．また，／C は同じ読みの大文字と小文字を同一の文字として扱います．バイナリ・ファイルを扱うときには，／B スイッチ(バイト単位に比較される)を使います．

結果のレポートは，両ファイルともに不一致部分とその前後の一致部分各1行を付けて表示されます．スイッチ／n(省略値は3)は一致とみなす判断を行なうための行数の指定となっており，一致関係が崩れた後再開する場合の判定に用いられます．

ところで，format コマンドで／S スイッチを指定せず，かつシステム・ディスクからの diskcopy も行なっていないメディアは，そのままではシステムを起動する能力をもっていません．もしシステム・ディスクとして使用するメディアならば，他のファイルをコピーする前に

　　sys 〈ドライブ名〉：

を実行して，システム関係のファイルをコピーしておかなければなりません．このコマンドで対象になるファイルは，属性(アトリビュート)が“S”のものです．

ファイルの属性は，次のコマンドで表示，変更できます．

　　attrib ［｜±｜R］［｜±｜H］［〈ファイル名〉］

●表1．3　attribコマンドで表示される属性の記号と意味

記号	意　味
A	アーカイブ(通常のファイル)
D	ディレクトリ
V	ボリューム・ラベル
S	システム
H	不可視(隠しファイル)
R	読み出し専用

●図1．3　chkdskコマンドによる表示の内容

```
       ボリュム名    作成日時
d:vvvvv は yyyy-mm-dd  hh:mm に作成されました
       nnnn k バイト：全ディスク容量
         nn k バイト：n 個のシステム・ファイル
         nn k バイト：n 個のディレクトリ
       nnnn k バイト：n 個のユーザ・ファイル
      nnnnn k バイト：使用可能ディスク容量
```

ここで，ファイル名は指定しないと，ヘルプ画面が表示されます．属性Rは読み出し専用(書き込み不可)を示し，Hは隠しファイル(dir，typeなどで表示できず，delもできない)を表わします．＋はその属性のセットで，－は解除のときに使います．単に属性を参照するだけならば，ファイル名のみを指定します．表示される属性値は表1.3のとおりです．

ディスクのボリューム・ラベルもformat時に指定できますが，その内容を表示，変更するには，

vol ［〈ドライブ名〉：］［〈新ラベル〉／S］

コマンドを使います．ドライブ名の省略時には，カレント・ドライブが採用され，新ラベル省略時には，現在のラベルが表示されます．ラベルの変更には，／Sスイッチの指定が必要です．

システム運用時にディスクの現況を知りたいときは，

chkdsk ［〈ドライブ名〉：］［／A］［／V］

コマンドを利用します．これによって図1.3のような内容が表示されます．／Aスイッチはすべてのファイルについてセクタ範囲のレポートを追加し，／Vスイッチはファイル名だけ追加表示するのに使います．

ハードディスクの運用

ファイルに対する各コマンドは，ハードディスクについても使用できます．

Human68Kでハード・ディスクは，フロッピーに比べて単に容量が大きいディスクと考えればよいようになっています．

ただし，ファイルのバックアップの問題は，容量が大きいだけに簡単ではありません．とはいえ，大容量だからこそデータが失われたときの被害も深刻なので，高度にハードディスクに依存したシステムでは，バックアップなしでの運用は考えられません．

ビジネス機では，**ストリーマ**という一種の磁気テープ装置を使って対応していますが，X-68000のようなホビー機では，フロッピーを使って何とかしのぐようにしています．そしてこのために，なるべく少ない枚数ですむようにデータを圧縮する

copy 2　〈ハードディスク記述〉〈フロッピーディスク記述〉

コマンドが用意されています．ここでハードディスク記述は，ドライブ全体からファイルまで任意のレベルを指定でき，フロッピーディスク記述は，ドライブ・レベルに限られます．

また，両者の記述順序を逆にすると，フロッピーにセーブされた内容を，ハードディスクに復帰する処理が行なわれます．

copy 2コマンドで作られたフロッピーの内容は，特殊な圧縮データのため他のコマンドで用いることができない点に注意が必要です．

1-9 ファイルのダンプ

❖関連コマンド　dump

実行形成式の機械語ファイルの内容を点検するには，16進値などのダンプをとらなければなりません．機械語ファイルに限らず，ファイル内容の「見えない文字」まで確認する場合には，ダンプが必要になります．

その際のコマンドは，

dump 〈ファイル名〉[／ [X] 〈開始アドレス〉[, 〈バイト数〉]]

を使います．ここで，開始アドレス，バイト数はファイルの先頭を0とした相対値とダンプする範囲の指定です．バイト数を省略するとファイルの最後まで，開始アドレスも省略するとファイルの全部がダンプされます．

スイッチの指定は，ダンプ内容の種類を減らすのに使われ，省略すると16進と文字，“／X”で16進のみ，“／”だけなら文字のみに限定されます．

ダンプ・リストの例を図1.4に示します．

●図1．4　ファイル・ダンプ・リストの例

```
A>dump batcmd.bat

00000000  65 63 68 6F 20 4F 46 46 0D 0A 3A 6C 6F 6F 70 0D echo OFF..:loop.
00000010  0A 20 20 20 20 69 66 20 25 31 20 3D 3D 20 6C 69 .    if %1 == li
00000020  73 74 20 67 6F 74 6F 20 6C 69 73 74 0D 0A 20 20 st goto list..  
00000030  20 20 69 66 20 25 31 20 3D 3D 20 45 20 67 6F 74   if %1 == E got
00000040  6F 20 65 78 69 74 5F 0D 0A 20 20 20 20 0D 0A 20 o exit_..    .. 
00000050  20 20 20 20 20 20 20 65 63 68 6F 20 4F 4E 0D 0A        echo ON..
00000060  20 20 20 20 20 20 20 20 72 65 6D 20 4D 65 73 73         rem Mess
00000070  61 67 65 20 25 31 0D 0A 20 20 20 20 20 20 20 20 age %1..        
00000080  65 63 68 6F 20 4F 46 46 0D 0A 20 20 20 20 20 20 echo OFF..      
00000090  20 20 67 6F 74 6F 20 6E 65 78 74 0D 0A 3A 6C 69   goto next..:li
000000A0  73 74 0D 0A 20 20 20 20 69 66 20 45 58 49 53 54 st..    if EXIST
000000B0  20 66 69 6C 65 31 20 74 79 70 65 20 66 69 6C 65  file1 type file
000000C0  31 0D 0A 3A 6E 65 78 74 0D 0A 20 20 20 20 73 68 1..:next..    sh
000000D0  69 66 74 0D 0A 20 20 20 20 69 66 20 45 52 52 4F ift..    if ERRO
000000E0  52 4C 45 56 45 4C 20 31 20 67 6F 74 6F 20 65 78 RLEVEL 1 goto ex
000000F0  69 74 5F 0D 0A 20 20 20 20 67 6F 74 6F 20 6C 6F it_..    goto lo
00000100  6F 70 0D 0A 3A 65 78 69 74 5F 1A                op..:exit_.
```

第2章

コマンド入力と自動実行

2-1 CTRL による特殊操作

❖関連コマンド　　break

Human68K では，表2.1のような特殊操作機能が用意されています．

この中で，CTRL+Cは処理中のプログラムをアボート(中途終了)させるのによく使われますが，ディスク・ドライブのイジェクト(メディアの排出)などにも使える便利な機能の1つです．

●表2．1　Human68K の CTRL による特殊操作一覧

キー入力	機能
CTRL + F1	ドライブAのイジェクトまたはハードディスクのOFF
CTRL + F2	ドライブBのイジェクトまたはハードディスクのOFF
CTRL + F3	ドライブCのイジェクトまたはハードディスクのOFF
CTRL + F4	ドライブDのイジェクトまたはハードディスクのOFF
CTRL + F5	ドライブEのイジェクトまたはハードディスクのOFF
CTRL + F7	ファンクション・キーの表示の変更
CTRL + OPT.1 + DEL	システムのリセット
CTRL + C	実行中のコマンドの中止
CTRL + H	コマンド行から最後の文字を削除（BSと同様）
CTRL + P +	画面表示をプリンタにも出力（一度押すとプリンタ出力の設定，もう一度押すと出力の解除）
CTRL + N	プリンタへの出力解除
CTRL + S	画面表示，スクロールの一時停止．任意キーで解除

Human68K を効果的に利用するコツは，的確な入力操作と，バッチファイルによる連続的な自動実行機能を上手に使うことにあります．

本章ではこの点に着目して，特殊キーなどを利用した入力方法(ヒストリ機能，テンプレート機能)を紹介し，バッチファイルの作り方についても実例をあげて説明します．

これらの操作機能は，常に使えるとは限らず，プログラム中に同じキー入力を別な目的で使用しているときは作動しません．たとえば，CTRL+Cをエディタではスクリーンに次ページの内容を表示させるのに使っているので，そのときエディタをアボートさせることはできません．

また，通常のプログラムでも，break コマンド(CTRL+Cの常時有効の可否を ON／OFF で指定)で OFF 指定になっている間(普通はこの状態)は，コンソール入力またはプリンタ動作以外にアボートできません．受け付けられる状態にあっても，誤って別なキーを押してしまった場合，そのキー入力の処理が終わらないと次のキー入力の処理に移れないため，TYPE のようにディスクからスクリーン表示を行なうだけのパターンのときなどは，一度誤入力したらアボートできなくなってしまいます．

同じことはCTRL+Sについてもいえます．CTRLを押し忘れるなどの誤入力のあと，正しく入力してももはやスクリーン表示を止めることはできません．この場合，TYPE などが終了した後コマンド画面に戻るとともに受け付けられ，コマンド・エラーとなるだけです．

2-2 特殊キー，ファンクション・キーの設定

❖関連コマンド　key

特殊キーやファンクション･キーは，メーカー出荷時に特定の意味をもっていますが，F 1～F32に対応するキーに限り，その制御内容が変更できます．参考までに，F11～F20は，F 1～F10とSHIFTを同時に押すことによって得られます．またF21～F32は，ROLL UPなどのキーと直接対応します．これらの関係は表2.2のとおりです．

これらのキーと制御情報との対応づけは，キー設定ファイル(通常 KEY.SYS)で情報をもち，その内容をシステムに登録することによって使えるようになるという関係にあります．

登録内容は**コマンド**や**ESC シーケンス**などがおもなもので，必要によってはパス名など任意の文字列を定義することもできます．ただし字数はF 1～F20が32字まで，そのほかは6字以内に制限されています．前者については別途画面に表示する文字列を＄FE に続き7字で定義でき，この場合は登録キー･データは24字以内となります．そのままでは入力できない特殊文字は，

・↵ …………CTRL＋ろ(カナ文字)
・BS …………CTRL＋む(カナ文字)
・＄FE ………CTRL＋へ(カナ文字)

の対応づけに従って入力します．なおESCはそのまま入力できます．

キー設定ファイルの更新，システムへの登録は，

Key

コマンドによって行ない，最初の問い合わせ

●表2．2　操作キーとファンクション・キー番号との対応表

キー	キー番号
F・1 ∫ F・10	F1 ∫ F10
SHIFT＋F・1 ∫ SHIFT＋F・10	F11 ∫ F20
ROLL UP	F21
ROLL DOWN	F22
INS	F23
DEL	F24
↑	F25
←	F26
→	F27
↓	F28
CLR	F29
HELP	F30
HOME	F31
UNDO	F32

更新ですか，登録ですか？［U／L］

に対し，更新ならばU，登録ならばLを選択します．あとは問い合わせに従って必要な内容を入力すればよいのですが，ファイル名の問い合わせに対してはKEY.SYSが省略値となります．

ESCシーケンスについては，次節「テンプレート機能とヒストリ機能」を参照してください．

2-3 テンプレート機能とヒストリ機能

❖関連コマンド　key

Human68Kでは，直前のコマンド入力内容を直接または加工して使えるように，**テンプレート機能**をもっています．また，直前に限らず入力されたコマンドはメモリに記憶されており，その内容を取り出してテンプレートに入れて活用できます．

テンプレート機能で記憶される内容は，↵を押したときに転送されます．↵を押さない限りコマンドに入力途中の内容が重ね書きされることはありません．

テンプレートに入った内容を参照する方法は表2.3のとおりですが，筆者の場合最もよく使うのはF 3です．これは現在の入力位置から後方に，対応位置のテンプレートの内容をコピーするもので，修正すべき

●表2．3　テンプレート機能一覧

キー入力	表示	機能
F・1 または → または ESC・S または	C1	テンプレートから現在行へ，1文字コピーする (COPY 1 CHARACTER)
F・2 または ESC・T（文字指定あり）	CU	テンプレートから現在へ，指定された文字の直前までのすべての文字をコピーする (COPY UP TO CHARACTER)
F・3 または ↓ または ESC・U	CA	テンプレートにあるすべての文字を現在行にすべてコピーする (COPY ALL CHARACTER)
F・4 または DEL または ESC・V	S1	テンプレート内の1文字をスキップする（コピーはしない） (SKIP 1 CHARACTER)
F・5 または ESC・W（文定指定あり）	SU	指定された文字の直前までテンプレート内の文字をスキップする（コピーはしない） (SKIP UP TO CHARACTER)
F・6 または HOME または ESC・E	VOID	現在行に入力した内容を取り消す（テンプレートの内容はそのまま）
F・7 または ↑ または ESC・J	NWL	現在行に入力した内容をテンプレートにコピーする
F・8 または INS または ESC・P	INS	挿入モードの設定・解除をする
F・9 または ESC・F	N & CU	現在行に入力した内容をテンプレートにコピーした後，指定された文字の直前までのすべての文字を現在行へコピーする
F・10	EOF	CTRL ＋ Z を入力する

●表2．4　ヒストリ行テンプレート間の操作キー

キー入力	機能
−〈数　値〉 [↵]	注目行から逆方向（古い方向）へ〈数値〉だけ移動し，その内容をテンプレートに格納する．〈数値〉を省略すると，1を指定したものとみなされる．
＋〈数　値〉 [↵]	注目行から正方向（新しい方向）へ〈数値〉だけ移動し，その内容をテンプレートに格納する．〈数値〉を省略すると，1を指定したものとみなされる．
−/〈文字列〉 [↵]	注目行から逆方向にある〈文字列〉で始まる行に移動し，その内容をテンプレートに格納する．
＋/〈文字列〉 [↵]	注目行から正方向にある〈文字列〉で始まる行に移動し，その内容をテンプレートに格納する．
−?〈文字列〉 [↵]	注目行から逆方向にある〈文字列〉を含む行に移動し，その内容をテンプレートに格納する．
＋?〈文字列〉 [↵]	注目行から正方向にある〈文字列〉を含む行に移動し，その内容をテンプレートに格納する．
[↵]	最新の行に移動し，その内容をテンプレートに格納する．
[ROLL DOWN]	注目行から逆方向へ1つだけ移動し，その内容をテンプレートに格納する．
[ROLL UP]	注目行から正方向へ1つだけ移動し，その内容をテンプレートに格納する．
[UNDO]	現在のテンプレート中の内容を，そのままコマンド行として実行する．[F・3] [↵] と押したのと効果は同じ．

ところは[BS]で戻して重ね書きし，削除は[DEL]，挿入は[INS]を点灯させて対処します．

他のキーも1字ずつコピーする([F 1])などいろいろ細かな機能をもっていますが，いちいち覚えるのも面倒で，またよほど長いコマンド行でない限り，修正手続きに時間をかけるメリットはありません．したがって，表を参照する時間があれば直接入力したほうが早いということになって，まじめにこの機能のすべてを活用しようとする気持ちにはなれないのです．

むしろ，**ヒストリ機能**のほうが活用するメリットが大きいと思われます．たとえば，何かのコマンドがうまくいかなかった場合，救済するため別のコマンドを働かせると，当初目的としたコマンドはもうテンプレートに残っていません．そこで，

　　his [↵]

コマンドを働かせると，その時点までのコマンドの履歴がスクリーンに表示されます．このコマンドは表2.4の相対位置でバックする機能(− 〈数値〉)などを使うときには実行しておいたほうがよいのですが，数行バックする程度ならば，直接[ROLL DOWN]を繰り返し押すのが最も速いでしょう．このとき，バックのしすぎは，[ROLL UP]で戻せます．そして必要なコマンドがみつかったら，その内容はテンプレートに入っているので，[UNDO]を押すと再実行されます．これら3つのキーは横に並んでいて覚えやすく，他のキーについて知らなくても，これだけでほとんどのケースに対応できます．

もし戻り先がかなり前ならば，hisコマンドで履歴表示を行なってスクリーンに表示された逆順の行番号(最後に投入したhisコマンドが相対0番となっている)を手掛かりに

　　−〈行番号〉

と入力すれば，その行番号の内容をテンプレートに入れることができます．この場合も，再実行させるには[F 3] [↵]か[UNDO]によらなければなりません．

ヒストリ機能はあくまで限られたメモリ・サイズのバッファを利用しているため，その範囲内でしか履歴を残すことができません．バッファ・サイズはCOMMANDコマンドの“／Hスイッチ”で指定するようになっているので，不足する場合はCONFIG.SYSのSHELL記述を直して対応することになります．

ただし，hisコマンドで表示してみて不足に気付き，あわてて同ファイルの手当を行なっても，それが有効になるのは次の立ち上げからです．消えてしまったものはどうしようもないので，再度コマンドを入力するしかありません．

ヒストリ機能がもっと光ってくるのは，**バッチ・ファイル**を作成する場合です．このときは，

his　**／B <バッチ・ファイル名>**

の形式で／Bスイッチを指定します．リダイレクトされてできるバッチ・ファイルには，行番号が削除されたものが入ります．あとはエディタで不要な行を削除したり，不都合な部分を修正するなどして仕上げをします．

このあたりの扱いについては，第5部のサンプル・プログラムの中でも取り上げているので参考にしてください．

なお，hisコマンドでは，表示する行番号の範囲を“,”で区切って指定し，限定できます．しかし，目的のコマンド群の範囲は，目見当で決めるか，直視できる画面に残っている部分をカウントするしかありません．正確を期すため別途hisコマンドで全部表示して調べるというのでは，笑い話になってしまいます．

2-4 プロンプトの変更

❖関連コマンド　prompt

Human68Kのコマンド画面では，普通，カレント・ドライブ名に続いて，“＞”がプロンプト(入力促進文字列)として表示されます．

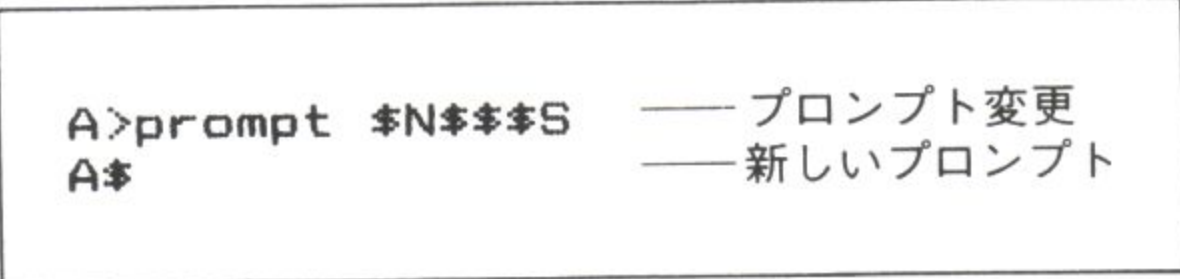
このプロンプトは任意に変更できるようになっており，新しいプロンプトは，

prompt　[**<新プロンプト>**]

●表2．5　特殊文字，変数の代入子

代入子	意味
$N	カレント・ドライブ名(大文字表示)
$n	カレント・ドライブ名(小文字表示)
$D	現在の日付
$T	現在の時刻
$P	カレント・ドライブ名とカレント・ディレクトリ名
$V	バージョン表示
$$	“$”文字
$G	“>”文字
$L	“<”文字
$B	“｜”文字
$−	改行
$S	ブランク
$H	バック・スペース
$E	ASCIIコード“$1B”(エスケープ・コード)

●図2．1　プロンプト変更の例
(カレント・ドライブ＋“$”＋スペースにする)

```
A>prompt $N$$$S     ——プロンプト変更
A$                  ——新しいプロンプト
```

で設定できます．たとえば

prompt Command＝

とすれば，次回からプロンプトは，

Command＝

に変わります．

このとき固定文字列以外に，特殊文字や変数についても，表2.5の代入子を作って記述できます．図2.1は，カレント・ドライブ，$，スペースの3文字からなるプロンプトを作る例で，prompt コマンド実行後，プロンプトはただちに指定されたものに変わります．この場合スペースを含んでいるので，dir などのコマンドを入力すると，

A $ ␣ dir　(␣はスペース)

のように続きます．

prompt コマンドで新プロンプトを省略すると，標準のプロンプトに戻ります．

2-5 バッチ・ファイルによる自動実行

エディタを使って，コマンド記述を並べたファイルを作り，拡張子“.bat”を付けると，バッチ・ファイルができます．そして，そのファイル名(拡張子は省略)をコマンド代わりに入力すると，ファイルに書かれているコマンド行が順番に実行される点がバッチ・ファイルの便利なところで，定型的な処理を自動化する際に大きな威力を発揮します．

バッチ・ファイルするにはコマンドをそのまま並べればよいのですが，一部分を文字変数にすると汎用的に使えるようにする場合には，変数部分を％1のように仮記述しておきます．％1はそのバッチ・ファイルから起動するときの第1パラメータに対応するもので，同様に，2番目のパラメータは％2のように書いておきます．こうしておいて，ファイル名で起動する際，コマンド行にパラメータを並べると，その値で％記述の内容が置き換えられ，実行されます．

図2.2は，Cによってプログラムを開発する際，エディタ(ed)，Cコンパイラ・ドライバ(cc)，不要ファイルの削除(del)，テスト実行を連続して処理する例です．パラメータは1つだけですが，このバッチ・ファイルはCプログラムのデバッグの際に大きな助けとなります．

テンプレート機能は，キーボードから入力したコマンドについて働くもので，一度バッチ・ファイルの処理を起動すると，終了後[F 3][↵]で何度でも繰り返して起動できます．上記の例の場合，この方法でデバッグがはかどるという寸法です．

●図2．2　バッチ・ファイルによるCプログラム修正～コンパイル～テスト実行の自動化

```
A>type at.bat   ┐
ed %1.c         │
cc %1.c         │ バッチ・ファイルの内容
del %1.s        │
del %1.o        │
%1              ┘
A>at abc        ─ このように起動すると
A>ed abc.c      ┐
A>cc abc.c      │ パラメータによって置き換えられた
A>del abc.s     │ 各コマンド行が実行される
A>del abc.o     │
A>abc           ┘
```

2-6 自動立上げ

Human68K では，一種のバッチ・ファイル AUTOEXEC.BAT が存在すると，システムの立ち上げ最終段階でこのファイルの内容が自動的に実行されます．したがって，この中に，

```
cd user
dir
```

のようなコマンドを登録しておけば，カレント・ディレクトリが“user”となり，そのディレクトリが表示されて起動されます．さらに進んで，

```
ed abc.c
```

のように，エディタを起動するコマンドを加えれば，立ち上げと同時にエディタの画面になります．

AUTOEXEC.BAT の内容は，エディタによって書き換えができるので，デバッグ中のときなどに任意のディレクトリへの cd コマンドやエディタの起動などを登録しておけば，起動と同時に作業にかかることができます．このファイルの内容を必要に応じて常時書き換えるのがシステムの上手な運用方法でもあります．

参考までに，メーカー提供の同ファイルの内容を見ると，path コマンドで外部コマンドの実行範囲を規定し，temp コマンドで作業用ディレクトリを設定したのち，ライブラリ・パスとインクルード・パスを set コマンドで与えています．これらの内容は，削除するとあとで不都合を生ずるものもあり，ユーザが追加する内容はあとに続けるのが最も無難です．

なお，AUTOEXEC.BAT ファイルは，ドライブ直属のファイルである必要があり，いずれかのディレクトリ下に所属する状態では，システム起動時に検索できません．

2-7 環境文字列の設定と参照

❖関連コマンド　　set

環境文字列は，システムで記憶している文字列変数で，

set [〈変数名〉= [〈文字列〉]]

でその値をセットできます．

そしてこの変数値は，コマンド中で，

%〈変数名〉%

形式により参照できます．すなわち，この形式の記述があると，環境文字列群の中から該当名のものが検索され，対応する文字列がその位置に代入されます．この方法で，ずっと離れた(後方にある)コマンドに変数を伝達することができます．

setコマンドは，パラメータを全部省略すると，現在時点の全環境文字列が表示されます．また，“＝”までで↵を押すと，その変数が消去されます．

環境文字列の内容は，通常のメモリによって記憶されているため，システム立ち上げ時点では存在していません．したがって，一般にはAUTOEXEC.BATファイルにsetコマンドを連ねて保持する方法が採用されています．

環境文字列は，一般にパス記述を肩代わりすることが多く，ユーザ定義のものはバッチ・ファイル中で参照するのがほとんどです．しかし，プログラム中で参照することも可能なため，グローバルなパラメータとして，複数のプログラムで利用できます．そして，setコマンドで任意の値に変更できるため，手動で変更して一時的な値で運用できる便利さを兼ね備えています．しかもAUTOEXEC.BATで次のシステムを立ち上げるときにはもとの値に戻せるので，平常処理，臨時処理といったパターンにも対応できます．

2-8 バッチ処理の制御

❖関連コマンド **goto, :, if, for, pause, rem, echo, shift**

バッチ処理は，前もって登録されたコマンド群を順次実行していくものですから，そのままでは柔軟性がなく，ワンパターンの処理しかできません．しかし，以下で紹介するバッチ処理制御コマンドを利用すると，条件によって異なる処理をするといったプログラムのような機能をもたせることができます．

このことを最も象徴的に示しているのは，

```
go to 〈ラベル〉
```

コマンドでしょう．このコマンドは，

```
: 〈ラベル名〉
```

形式で定義されたラベル行に飛ぶ働きをもっており，あたかもBASICのGO TO文のように使えます．飛び先は必ずしも下ばかりとは限らず，上の行にバックしてもかまいません．

条件判断は，

```
           | 〈文字列1〉 == 〈文字列2〉 |
           |            〈数値〉        |
if [not]   | ERROR LEVEL 〈数値〉       |  〈コマンド〉
           | EXIT CODE 〈数値〉         |
           | EXIST 〈ファイル名〉       |
```

によって行ない，ここでコマンドにgotoを使えば，条件が成立したとき任意のラベルに飛ばすことができます．成立条件は，文字列の等式では，たとえばパラメータの値を調べるようなときに使われ，一致したとき成立とみなされます．数置の場合は0でないとき，ERROR LEVELはCOMMAND.Xからのエラー・コードが数値以上の場合，またEXIT CODEは等しい場合に成立します．EXISTはファイルが存在するとき成立として扱われます．

これらの条件の前には，notを付けることができます．このとき，条件の成立関係が逆になり，notを付けない状態で不成立だったものが成立として扱われます．

処理の制御を行なうコマンドには，

```
for [%] %〈変数名〉 in 〈項目リスト〉 do 〈コマンド〉
```

もあります．このコマンドの機能は，繰り返し実行する点ではBASICと似ていますが，繰り返す回数は項目リストの数によって決まります．すなわち，項目リストの内容が1つずつ変数に代入され，その状態でコマンドが1回ずつ実行されるのです．変数名の前の%の数は，直接コマンドとして使うときは1個，バッチ・ファイルの中では2個付けます．また，変数名はパラメータと区別するため，数字は使わないようにします．

バッチ・ファイルから連続的に実行する場合，プリンタのセット待ちなど，一時停止したいことがあります．このようなときは，

```
pause [〈コメント〉]
```

コマンドで，指定があればコメントを表示して，待ちに入らせることができます．待ちを解除するには，何か適当なキーを(たとえばスペースなど)を押せば再開されます．

停止せずにコメントだけ表示するには，

●図2．3　制御コマンドを使ったバッチ・ファイルの内容と実行例

```
A>type file1                              ┐
**            **                          │ file1の内容
*     file1     *                         │
**            **                          ┘
A>type batcmd.bat                         ┐
echo OFF                                  │
:loop                                     │
    if "%1" == ""   goto ex_              │ バッチ・ファイルの
    if %1   == list goto list             │ 内容
    echo ON                               │
    rem Message %1                        │ (パラメータが"list"ならばfile1の
    echo OFF                              │  内容を表示する.
    goto next                             │  その他はパラメータそのものを表
:list                                     │  示するものとし，パラメータがな
    if EXIST file1 type file1             │  くなったら終了する.)
:next                                     │
    shift                                 │
    goto loop                             │
:ex_                                      ┘
A>batcmd ABC list 1234                    ┐
A>echo OFF                                │
A>    rem Message ABC                     │
A>    echo OFF                            │
**            **                          │ 実行結果
*     file1     *                         │
**            **                          │
                                          │
A>    rem Message 1234                    │
A>    echo OFF                            │
A>                                        ┘
```

rem　[〈コメント〉]

を使います．このコマンドは，現在何をしているかなどを操作者に知らせるのに使われます．

同じことは

echo　[〈コメント〉]

でも行なえます．ただし，このコマンドは，エコー・モードをON／OFFするのにも使われ，コメントの前の位置にいずれかを指定することができます．ONならばコマンド行が表示され，OFFならば表示を抑制します．通常はONですが，コマンド行の表示が多すぎたり，見えると支障がある場合などに，OFFにします．なお，同じコメントでも，remによるものは，OFFモードで表示されません．

バッチ・ファイル内で使用されるパラメータ記述は，％1～％10だけですが，

shift

コマンドが実行されると，実パラメータ(起動時のコマンドで与えられた内容)の先頭のものが消去され，以下のパラメータの順位が1つずつ繰り上がります．結果的に，10個以上の実パラメータを指定することができるようになるのです．

これらのコマンドの使用例を，図2.3に示します．

2-9 コマンド・プロセッサの起動と終了

❖関連コマンド　command, exit

Human68K の立ち上げの際，コマンド・プロセッサの COMMAND.X は，CONFIG.SYS の SHELL 行で起動されます．

その後も，入れ子の形で子プロセッサとしてコマンド・プロセッサを立ち上げることができ，工夫次第で面白い使い方ができます．

さて，コマンド・プロセッサの立ち上げは，

```
comnand  [／P] [／D] [／B: <バッチエリア・サイズ>] [／E: <環境エリア・サイズ>]
         [／H: <ヒストリエリア・サイズ>] [／C <コマンド>]
```

コマンドによって行ないます．ここで，各種のエリア・サイズを指定するスイッチは，デフォルト値から変更したいときに使用します．それぞれ1個当たり256バイトで，0～255個の範囲で指定できます．ただし0は2個を表わし，省略時はこの値がとられます．その他のスイッチについては，次のとおりです．

／P スイッチは，立ち上げたコマンド・プロセッサが一時的なものでない(終了させることができない)ことを宣言するものです．したがって，このスイッチを使うと，コマンド・プロセッサを終了させる

```
exit
```

コマンドが無効になります．また，このスイッチが指定されると，立ち上げ過程で AUTOEXEC.BAT ファイルが実行されます．この実行をバイパスしたいときは，さらに／D スイッチを追加指定します．

／C スイッチは，コマンド・プロセッサに指定コマンドを実行させるために臨時に起動するものです．この場合，“／C” を省略して，直接コマンドを記述することもできます．このような性格上，指定されたコマンドの実行がすんだら eixt コマンドなしに子プロセッサは終了します．

ところで，バッチ・ファイルの中から直接他のバッチ・ファイルの内容は実行できませんが，command コマンドを経由するとそれができます．このとき／C スイッチを指定し，入れ子となるバッチ・ファイルを呼び出せばよいのです．

第3章

その他の機能

3-1 起動モードを決めるCONFIG. SYSファイル

❖関連コマンド　custom

Human68K は立ち上げ時に複数のファイルを参照します．そのうちの１つが CONFIG.SYS で，このファイルはシステムの初期設定に用いられます．

ここで定義される内容は，次のとおりです．

・FILES……………同時オープンできるファイル数(５～93，省略時15)
・BUFFERS………ディスク I/O バッファの数(２～99，省略時20)
・BELL ……………警告用の PCM ファイル名(通常￥SYS￥BEEP. SYS)
・DEVICE …………システムに組み込むデバイス・ドライバ名
・BREAK ………… ON/OFF で，常時[CTRL]+[C](アボート)を有効にするかどうかを指定．OFF のときはコンソール入力時とプリンタ出力時のみ有効(省略時 OFF)
・SHELL ………… コマンド・プロセッサの起動コマンド．(省略時 COMMAND. X)

ユーザーがこの内容を変更するケースとして最も多いと思われるのは，デバイス・ドライバの増設です．あとはメモリ・サイズの調整でファイルやバッファ数を減らしたり，ヒストリ機能の増強のため COMMAND. X の/H パラメータを指定するなどの操作が考えられます．

変更時には，直接このファイルをエディタで修正するか，

```
custom
```

コマンドを用いて，各登録内容を対話式に入力する方法があります．後者では複数のデバイス・ドライバに対応するため，DEVICE について何も入力しないで，[↵]を押すまで繰り返し入力を求めてきます．また，

本章では，Human68K の機能およびコマンドのうち，第1章，第2章で取り上げなかったものについて説明します．

分類としては，いわば「雑コマンド」に属するものが多いのですが，システム立ち上げ時に参照される CONFIG. SYS ファイルの作り方やフィルタなど有用な機能も紹介しており，コマンドをマスターする上での「総仕上げ」というイメージで読んでいただければ幸いです．

省略値を入力したいときは，[F 3]で代替する機能があります．

CONFIG. SYS はシステム立ち上げ時にのみ参照されるので，変更した内容が有効になるのは次の立ち上げ時からです．

また，記述内容が誤っていると，立ち上げに失敗することになります．したがってそのシステム・ファイルを使用する限りはエディタも動かず，CONFIG. SYS の内容を訂正することさえできなくなります．このように，バックアップをとらずに変更することは大変危険なので，注意が必要です．

なお，ビジュアル・シェルが起動されるのは，SHELL 行で VS. X が指定されたとき，または TITLE. SYS ファイルが存在して SHELL 行の指定が省略されたときに限られます．

3-2 フィルタの活用

❖関連コマンド **more, pr, find, sort**

おもにパイプラインによって使われ，データの流通をコントロールする種類のコマンドを**フィルタ**といいます．

たとえば，type コマンドのスクリーン表示はいわば「タレ流し」で，長いテキストの場合，そのままでは[CTRL]+[S]によって適当な位置で止めながら見ないと使いものになりません．このようなとき，ページご

とに停止するフィルタ more を,

type 〈ファイル名〉 | more

形式で使用すれば，各ページごとに静止した状態で参照することができます．実行中に次ページに進めたいときは，何か適当なキーを押してやります．

同様なことをプリンタに行ないたいときは，次のフィルタを使います．

pr [／WN] [／Ln] [／H 〈ヘッダ文字列〉] [／B 〈タブ・サイズ〉] [／F]
[／T] [／N]　[〈入力ファイル名〉] [〈出力ファイル名〉]

このコマンドでは，各スイッチで次のような指定が行なえます．

- ／Wn ……………行当たり文字数(n)．省略値：80
- ／Ln ……………ページ当たり行数(n)．省略値：66
- ／H 〈ヘッダ〉 …各ページの頭書きの内容．省略値：ファイル名と同じ
- ／B 〈タブ〉 ……水平タブのサイズ(2，4，8，16)．省略値：8
- ／F ………………ページの終わりにFF(フォーム・フィード)を付ける
- ／T ………………頭書きの省略
- ／N ………………各行に行番号を付ける

ファイル名は並記されたとき右側のものが出力ファイル名として扱われ，1個だけのときは入力ファイルとみなされます．両方とも省略すると，パイプ動作ができます．もしパイプ動作時に prn などプリンタに出力したいときは，標準出力をリダイレクトする方法があります．

「フィルタ」という語を最もよく実感させるコマンドとしては，

find [／V] [／C] [／N] [／L] [／F] "〈文字列〉" [〈ファイル名〉]

があげられます．これは基本的には指定文字列を含む行のみを通過(出力がスクリーンのときは表示)させるもので，／V スイッチの指定があるとその反対の動作をします．また，／C スイッチは／V，／N，／F に対し排他的に働き，該当行数，非該当行数，全体の行数をカウントして報告します．比較の際大文字，小文字の区別をしないようにするには／L スイッチを使い，出力する行の先頭の行番号をカットするときは／N，ファイル名を先頭に付加するときは／F スイッチを指定します．

特殊なフィルタとしては，分類を行なう，

sort [／R] [／I] [／+n] [／T 〈タブサイズ〉]

●図3．1　sori. dat ファイルの内容

```
001Yoshida
002Katayama
003Ashida
004Yoshida
005Hatoyama
006Ishibashi
007Kishi
008Ikeda
009Sato
010Tanaka
011Miki
012Fukuda
013Ohira
014Suzuki
015Nakasone
016Takeshita
```

があります．このコマンドは，データ行の全体の内容を比較し，／R スイッチが指定されていないときは昇順(小→大)に，指定されているときは降順(大→小)に並べ替えます．比較の際は／+n スイッチが指定されていないときは行の先頭(+1に相当)から，指定されているときはその位置から行末にかけて行なわれます．また，／I スイッチは同じ読みの英字に対し大文字，小文字の区別をしないように働きます．すなわちこの指定がないと，小文字は大文字より大きいものとして扱われます．なお，データの中にタブが含まれているファイルについては，／T スイッチでタブサイズを指定しないと比較位置がズレる心配があります．

図3.1は，pr, sort コマンドをテストするために作成したデータ・ファイルの内容を示したものです．

このデータを，／L スイッチでページ当たり15行を指定して，pr コマンドによって印字した結集を図3.2に示します．

また，同じデータを使ってアルファベット順に分類した結果は，図3.3のとおりです．

●図3．2　sori. dat を pr コマンドで出力した例（ページ当たり15行）

```
001Yoshida
002Katayama
003Ashida
004Yoshida
005Hatoyama
006Ishibashi
007Kishi
008Ikeda
009Sato
010Tanaka
011Miki
012Fukuda
013Ohira
014Suzuki
015Nakasone
016Takeshita
```

●図3．3　sori. dat を sort コマンドでアルファベット順に分類した例

```
A>sort /+4 <sori.dat
003Ashida
012Fukuda
005Hatoyama
008Ikeda
006Ishibashi
002Katayama
007Kishi
011Miki
015Nakasone
013Ohira
009Sato
014Suzuki
016Takeshita
010Tanaka
001Yoshida
004Yoshida
```

3-3 スクリーンの制御

❖関連コマンド　screen, cls

Human68K のスクリーンは，テキスト行とグラフィックの両方に対応できるようになっています．

スクリーンの表示は，縦方向に512本の輝線，横方向に768または512ドットから構成されており，そのうちのどちらのドット密度で表示するかを選択できます．この違いは，テキスト画面では横96字にするか64字にするかという差に相当し，Human68K のコマンド画面では，普通96字モード採用しています．

グラフィックの場合はリフレッシュ・メモリをたくさん使い，この傾向は色数が多いほど強くなります．したがって，色数を変えるためには，メモリの再編成が必要になるのです．

こういった要素は，

```
screen  [[〈画面サイズ〉], [〈グラフィック・モード〉], [〈表示モード〉]]
```

コマンドで，表3.1のパラメータによって設定します．各パラメータは，全部省略するとすべて 0 が指定されたとみなされ，一部省略すると以前設定した値が省略値となります．グラフィックのリフレッシュ・メモリ領域を RAM ディスクに使っているときは，当然ながらグラフィック・モードは 0 以外にすると衝突するので注意が必要です．

このほかのスクリーンの制御コマンドとしては，画面クリアの

```
cls
```

があります．これには，クリアとともにカーソルをホーム・ポジションに戻す機能が加わっています．

●表 3．1　screen コマンドの設定値

パラメータ	設定置	意　味
画面サイズ	0	横96文字（グラフィック768×512ドット）モード
	1	横64文字（グラフィック512×512ドット）モード
グラフィック・モード	0	グラフィックなし
	1	グラフィック16色
	2	グラフィック256色（〈画面サイズ〉が 0 のときは無効）
	3	グラフィック65,536色（〈画面サイズ〉が 0 のときは無効）
表示モード	0	テキストのみ表示
	1	テキスト，グラフィックを表示（〈グラフィック・モード〉が 0 のときは無効）
	2	テキスト，スプライトを表示
	3	テキスト，グラフィック，スプライトを表示（〈グラフィック・モード〉が 0 のときは無効）

3-4 RS-232Cポート(補助入力デバイス)の制御

❖関連コマンド　speed, ctty

Human68K では，コンソール(操作卓)機能を標準のキーボードとスクリーンから，RS-232C ポートに切り換えることができます．このため RS-232C ポートは，「補助入力デバイス」と呼ばれています．

RS-232C ポートは，接続先のデバイスによって**ボーレート**(転送速度)その他が異なるため，動作を開始する前に個々のパラメータを設定してやらなければなりません．このための方法には２つあって，１つは再立ち上げの前まで有効な

```
speed [〈パラメータ〉]
```

コマンドを利用するやり方です．speed コマンドのパラメータには表3.2のものがあり，コマンド起動時に指定しないときは，現在の内容が表示された後，プロンプト(−)に従って入力が要求されます．これを省略して ⏎ だけで終わらせると，パラメータの内容は変更されません．

もう１つの方法は，switch コマンドによるもので，次回の立ち上げ時から有効になります．これについては，同コマンドの説明を参照してください．

コンソールの切り換えは，

```
ctty [ | AUX | ]
       | CON |
```

コマンドで行ないます．ここで，主コンソールから補助コンソールに変更するときは **AUX**，反対の場合は **CON** を指定します．

●表３．２　speed コマンドのパラメータ一覧

パラメータ	指定内容
ボーレート (転送速度)	9600 4800 2400 1200 600 300 150 75
キャラクタ長 (１字のビット数)	BITS-8(８ビット)〔B８〕 BITS-7(７ビット)〔B７〕 BITS-6(６ビット)〔B６〕 BITS-5(５ビット)〔B５〕
パリティ・チェック	PARITY-NONE　(パリティなし)〔PN〕 PARITY-EVEN　(偶数パリティ)〔PE〕 PARITY-ODD　(奇数パリティ)〔PO〕
ストップ・ビット数	STOP-１ (ストップ・ビット1)〔S１〕 STOP-２ (ストップ・ビット2)〔S２〕
XON，XOFF の指定	XON　(バッファあふれ制御あり) NONE (バッファあふれ制御なし)

●〔　〕内は短縮形

3-5 メモリ・スイッチの設定と内容表示

❖関連コマンド

swith

パソコンでは以前，システム立ち上げ時のモード設定などにDIPスイッチを使っていましたが，バッテリ・バックアップ式のRAMが普及するようになって，メモリとして記憶する方向に改善されてきました．

このようにするほうがユーザーにとって設定内容がわかりやすく，またメーカーとしては簡単に項目を増やせます．

立ち上げパラメータの内容は，表3.3のとおりです．参照および変更時のコマンドは，

switch　{[<パラメータ名>= <指定内容>]}

を使い，パラメータを省略すると，個々の現状が表示がされてから変更入力を受け付けるプロンプト(－)が表示されます．このとき，参照だけの場合は，↵を押せば終了します．

これらの内容は，あくまで立ち上げ時にシステムの初期設定に使われるもので，変更してもすぐに有効にはなりません．ただちに効果が出て欲しいときには，RS-232Cパラメータの場合に限り，speedコマンドで同じことを設定する方法があります．いずれのパラメータも，再立ち上げすればすぐに有効になります．

●表3．3　switchコマンドのパラメーター覧

パラメータ名	指　定　内　容
RS-232C　〔R〕	表3.2参照（複数あるときはスペースで区切る）
MEMORY　〔M〕	nnnnK（標準：1,024K）
BOOT　〔B〕 （システム起動ドライブ）	STD（FD0～HD15の順に調べて起動） 2HD0～3(指定されたFDから) HD0～15（指定されたHDから） ROM$nnnn（ROMの指定アドレスから） RAM$nnnn（RAMの指定アドレスから）
EJECT　〔E〕 （電源OFFイジェクト）	ON（電源OFFのときイジェクトされる） OFF（電源OFFでもイジェクトされない）
OPT.2KEY　〔O〕 （TVコントロール）	TVCTL（OPT.2キーでTVコントロールする） NORMAL（TVコントロールしない）
CONTRAST　〔C〕 （輝　度）	nn（0～15で0は無表示．15は輝度最高）
KANA　〔K〕 （カナキーの配列）	JIS（キーボード上の表記どおり） AIU（50音順）

3-6 日時その他の設定および表示

システムのいろいろな設定項目のほとんどは，すでに説明しましたが，ここでは残りのものについてまとめて取り上げます．

●システムのバージョン表示

Human68K のバージョンを表示したいとき，

```
ver
```

コマンドを投入すると，

```
Human68K version1.00
```

のように表示されます．

●ディスク書き込みベリファイ機能の設定と解除

ディスク書き込み時の読み出しによる確認チェックを行なうかどうかの指定は，

```
verify [| ON  |]
        | OFF |
```

により，チェックする(ON)か，しない(OFF)かを指定します．通常は OFF になっています．

●日付の表示および設定

日付はクロックにより自動的に更新され，電源を切ってもバッテリ・バックアップによって動作が維持されます．そしてその表示および設定には，

```
date [〈年〉 — 〈月〉 — 〈日〉]
```

コマンドが使われます．カッコ内を省略すると，表示後に変更入力を受け付けるので，表示のみのときは ↵ でバイパスします．変更時には，年は4桁，月，日はそれぞれ2桁で入力しますが，年は西暦の下2桁でもかまいません．また，月，日は上位が0のとき1桁で入力することができます．その代わり，間の“—”を忘れないようにしなければなりません．

●時刻の表示および設定

時刻も日付と同様に，クロックによりカウントされています．その設定，表示方法もよく似ており，

```
time [〈時〉: 〈分〉: 〈秒〉]
```

コマンドによって処理します．カッコ内の省略についても同様です．

とくに秒合わせをしたいときは，キーボードで入力している間にもどんどん経過していくので，たとえばラジオや TV の時報に合わせて待機するとか，秒合わせずみの時計を見て先回り入力をしておき，↵ キーを設定時刻に合わせて押すという方法が考えられます．

3-7 外字登録

❖関連コマンド　uskcgm

外字登録は，外字ファイルの更新(U)とシステムへの登録(L)の2段階によって行ないます．

uskcgm

コマンドを立ち上げると，どちらのモードかの問い合わせが行なわれるので，段階に応じてUかLのいずれかを入力します．続いてファイル名の入力が要求されますが，[↵]でスキップすると，“USKCG.SYS”が採用されます．登録ならばこの段階で終了です．

更新のときは引き続き出力ファイル名の要求があり，別ファイルを作る必要があれば名前の異なったファイル名を指定します．慣れないうちは，別ファイルを作るようにしたほうが無難と思われます．

次の段階は，コード番号に対応するドット・データの入力画面(図3.4)に移り，テン・キーでマスを埋めます．その際“0”キーはSET(白で埋める)，RESET(白を消去する)を切り換えるトグル動作(押すたびに反対の状態になること)となり，“5”キーはカーソル位置の状態を反転します．他のテン・キーは“0”キーで設定した状態に従って現在のカーソル位置をセットまたはリセットし，続いてそれぞれの方向(図3.5)に1つだけカーソル位置を移動します．セットやリセットを伴わず，単にカーソルだけ動かしたいときは，[↑]などのカーソル移動キーを使用します．

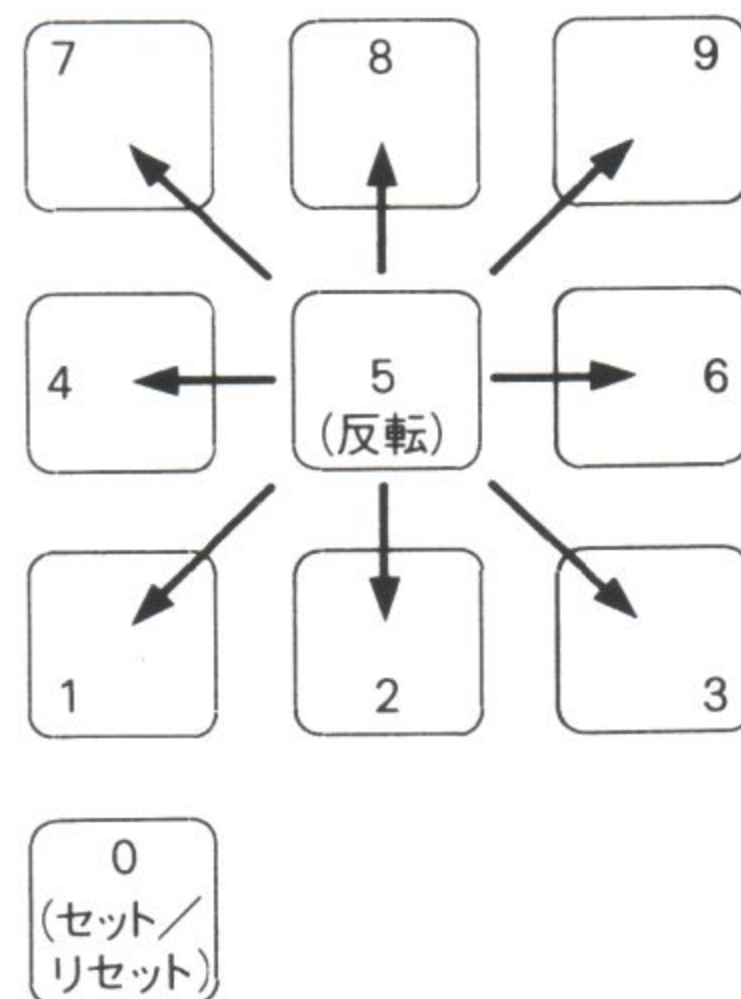
文字コードは[ROLL UP]で1つ後に進み，[ROLL DOWN]でバックします．作成した文字コード対応ドット・データは，[F 1]でファイルの書き出しをします．[F 2]は削除のとき使いますが，このときはコード番号の入力が要求されます．

入力されたドットは，マス目以外にもスクリーンの右上に実際の字体で表示されるので，直接結果を確認できます．また，ファイルの登録内容を一覧するときは，[F 3]を押すと，文字コードと該当文字が登録状態で一覧表として表示されます．表示ページの後退は，ROLL DOWN，前進はROLL UPを含む任意のキーを押します．一覧の中止は[ESC]によります．

編集したいコードが飛び飛びのときは，[F 4]を押して直接コードを指定する方法がベストです．よくあることですが，別の文字の登録内容を少し変更して使用したいときは，[F 6]で他の文字パターン(現在システムに登録されているもの)を呼び出せます．

●図3.4　外字の編集画面

```
入力ファイル名: USKCG.SYS          コード番号: 7621           [16x16]
出力ファイル名: USKCG.SYS          (7621-777E,F400-F5FF)

 0          : SET / RESET
 5          : ON / OFF
 1-9        : RESET & MOVE
 カーソル key : MOVE

 ROLLDOWN   : 前コード
 ROLLUP     : 後コード
 SPACE      : 16/24
 HOME       : HOME
 CLR        : CLR

F・1  :登録            F・6  :参照
F・2  :削除            F・7  :反転
F・3  :一覧            F・8  :回転
F・4  :コード入力
F・5  :ファイル読込    F・10:終了
```

●図3．5　テン・キーの機能

その他必要に応じて，[F 7]による白黒反転，[F 8]による90°回転なども使えます．[F 5]のファイル読み込みは，同じファイルまたは他のファイルを読み直すときに使われ，メモリの編集領域にかぶせる形で働きます．このため以前のメモリの内容は消えてしまうので，そのことを承知の上で使わなければなりません．

処理の終了は[F 10]を使い，ファイル更新の問い合わせに対し“Y”を入力すると，更新と同時にシステムへの登録も行なわれます．

第4章

エディタの使い方

4-1 スクリーン・エディタ(ED)の立ち上げ

スクリーン・エディタ(ed)を立ち上げるには，

ed {**[スイッチ]**} {**[ファイル名]**}

形式でコマンドを入力します．普通は，

ed　abc. s

のように，ファイルを1個だけ指定しますが，複数個指定して順次処理することもできます．

ファイル名は，アセンブラ・ソースなら“**.s**”，c ソースなら“**.c**”，BASIC ソースなら“**.bas**”(または“**.b**”)，バッチ・ファイルなら“**.bat**”の拡張子まで指定しなければなりません．一般のデータ・ファイルについては，拡張子は任意です(省略も可)．対象ファイルには，“CONFIG. SYS”も含むことができます．

ファイル名を省略すると，“ED. $ $ $”という仮の名前がとられ，いわゆる「名なしの権兵衛」扱いとなります．エディタの操作練習にはこのままで差し支えありませんが，プログラムのソースなどでは，上述のような拡張子付きの正式な名前に変更(ren コマンドを使用)してやらなければなりません．

ed スイッチは次のようなものがあります．

・**－S**……画面モード．1：96字／行，2：64字／行．省略時1．
・**－M**……行の最大長．128，256，512，1,024のいずれか．省略時512．
・**－M**……水平タブ間隔．2，4，6，8のいずれか．省略時8．
・**－T**……タブ記号(→……)の表示．省略時表示せず．
・**－L**……改行記号(↓)の表示．省略時表示せず．
・**－E**…… EOF(ファイルの終わり)の表示．省略時表示せず．

エディタはバッチ・ファイルや各言語のソース・プログラム・ファイルを作成する上で，なくてはならない重要なツールです．

しかし，マニュアルを見ると，ずい分たくさんの機能が書かれており，またキー操作にも重複があって，かなり複雑なのが実態です．

そこで，本章ではその中のよく使われる部分から少しずつ紹介し，最後に全部の関連を一覧表にして解明するという手順を採用しました．はじめてHuman68Kのエディタに触れる読者には手掛りを与え，また生かじりだった読者にはエディタの全体像が見えるようにするのが本章の目的です．

・－B ……バイナリ・ファイルの指定．省略時はキャラクタ・ファイル．

・－A ……ヘルプ・ファイルのパス．省略時はed. xのものと同じ．

この中で，バイナリ・ファイルとは実行形式のプログラム・ファイルなどが該当し，キャラクタ・ファイルのように，eof($1A)コードで終了させることができないため，ディレクトリ情報のファイル・サイズで終端を認識させるようになっているものです．ただし，実行形式のファイルをエディタで修正するのは困難で，一般の2進データ・ファイルに限られます．

4-2 最小限知っておきたい特殊キーの使い方

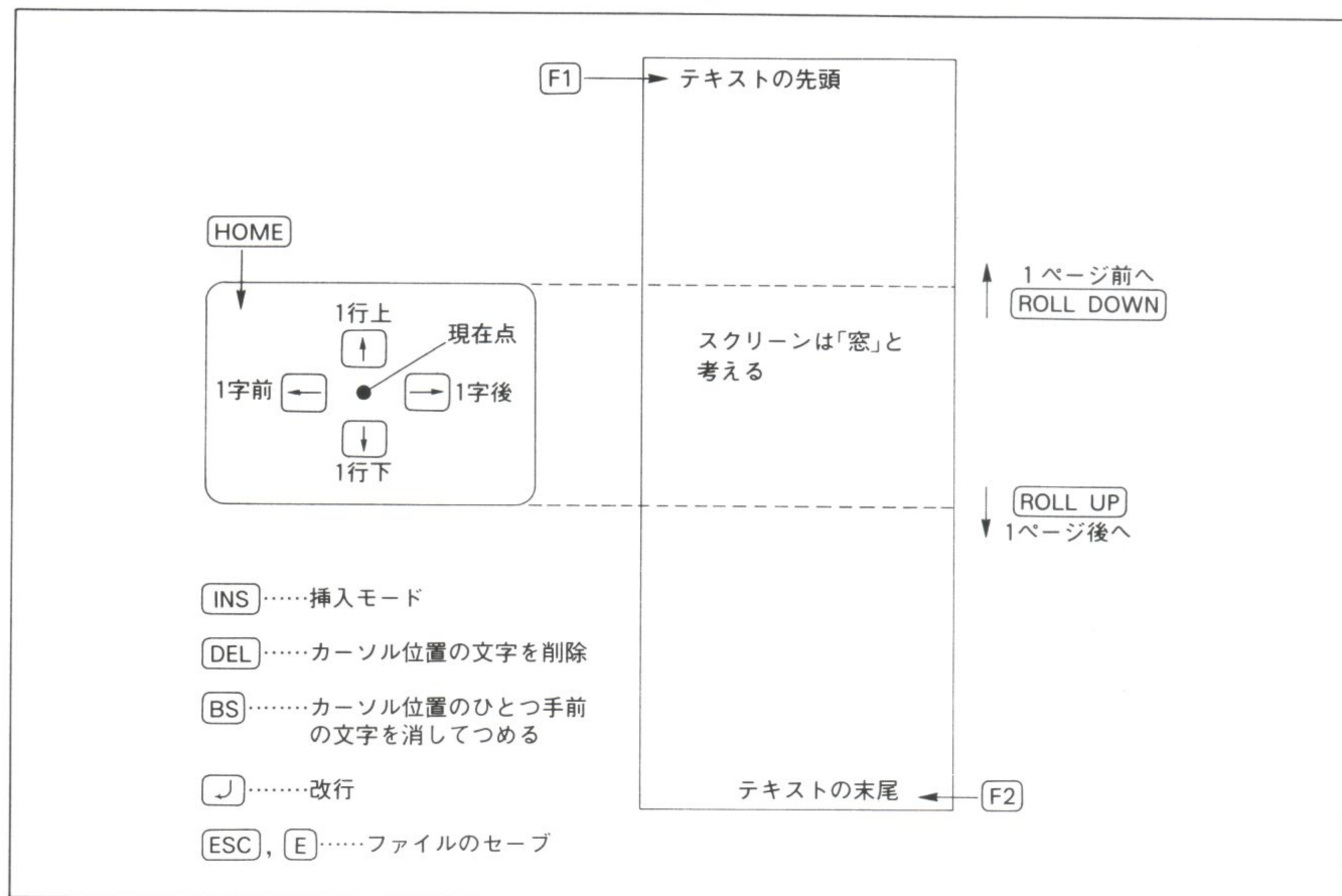
Human68Kのエディタは，他のいろいろなエディタの操作方法をミックスしているため，他機のエディタに慣れたユーザが移行する際には比較的抵抗が少ないという特徴があります．しかし，そのためにマニュアルの説明が多くなり，すぐに使いたいときに混乱が生じる点もない訳ではありません．

そこで本章では，初めてエディタを使うときに，最小限知っておいたほうがよいと思われる特殊キーに限定して説明を始めることにします．

図4.1は，エディタ動作時のテキストとスクリーンの関係を，特殊キーと対応づけして描いたものです．

エディタが起動されると，ファイルからテキストの内容がメモリにロードされ，スクリーン上に最初は，テキストの先頭から1ページ分のテキスト行が表示されます．このとき，INSキーはLEDが点灯した状態にあり，挿入モードになっています．このモードは，INSキーを押すたびにON・OFFの切り換えがなされます．

●図4．1　エディタ（ED）で知っておきたい最小限の特殊キー

F1 → テキストの先頭
HOME
1行上
現在点
1字前
1字後
1行下
1ページ前へ
ROLL DOWN
スクリーンは「窓」と考える
ROLL UP
1ページ後へ
INS……挿入モード
DEL……カーソル位置の文字を削除
BS………カーソル位置のひとつ手前の文字を消してつめる
↵………改行
ESC，E……ファイルのセーブ
テキストの末尾 ← F2

新しくテキストを書き始めるときは，テキスト内容を次々と記述していき，↵で改行するだけです．もし，任意のポイントに追記や削除を行ないたいときは，↑や←，→，↓によって該当位置にカーソルを移動させ，追記の場合にはそのまま挿入文字を打ち込みます．このときINSのLEDが点灯していないと重ね書きされ，以前の内容が消されてしまうので注意が必要です．

カーソル位置の文字を削除したいときは，DELを使います．行削除の方法については，次節で述べますが，当面はDELを押し続けることで急場はしのぐことができます．削除の方法としては，BSによってバックしながら消していくやり方もあります．このとき，カーソル位置の１字手前の文字が消去される点に注意が必要です．

↵は挿入モードのとき，行末の改行コードとなって１字分占有します．そして，その行では↵の位置から後にカーソルを移動させることができません．したがって↑や↓キーを押しても，まっすぐ上や下に移動するとは限りません．目視による移動予想位置が↵の後にあるときは，強制的に↵の位置に変更されます．↵は普通スクリーンで直接見えませんが，ED立ち上げ時に**－L**スイッチを指定すると，水色で“↓”が代わりに表示されます．

スクリーンは，あくまでテキスト全体から見れば「のぞき窓」のようなものです．この表示範囲をページと考え，前のページに切り換えたいときはROLL DOWN，後ろのページに移動したいときはROLL UPを使います．

もっと極端に，テキストの先頭まで戻したいときはF１，テキストの追加などで末尾まで送りたいときはF２を利用します．スクリーンの範囲内で先頭位置に戻したいときは，HOMEがその働きをします．

テキストの入力が終了したときや，ひとまず休憩のためファイルにセーブしたいときはESCを押し，次にEで終わらせます．ESCを押して操作するやり方を**ESCシーケンス**と言います．

ED終了後，更新前のファイルは，拡張子“**.BAK**”が付けられて保存されます．このとき，すでに“.BAK”付きのファイルがあれば，古いものが消されます．

なお，更新してしまってから前の状態に戻したいときは，更新後のファイルを消去して，“．BAK”付きファイルをリネーム（renコマンドを使う）すればよいのですが，ED終了時点で気付いたときは，ESCシー

ケンスでEの代わりにQを入力すれば更新後のファイルがセーブされず，結果として更新が行なわれません．

以上のキー操作を知っていれば，応急的にエディタを操作することができます．しかし，行単位での取り扱いとか，文字列の置き換えなど強力な(効率のよい)機能が外に用意されているので，あとでそれらについてもマスターしておいた方がよいでしょう．

なお，EDの操作には↑などの単独キーによるもの，ESCシーケンスによるもの，CTRL＋形式で操作するものの3通りがあり，全部覚えるのは大変なだけでなく無意味です．したがって，本書では単独キー操作を中心にして，やむを得ない場合に限ってその他の操作形式に言及することとします．

4-3 行の削除と移動，転写

行の削除や移動などを行なう際には，**カット・バッファ**というメモリの「入れ物」に対する操作キーを利用して行ないます．

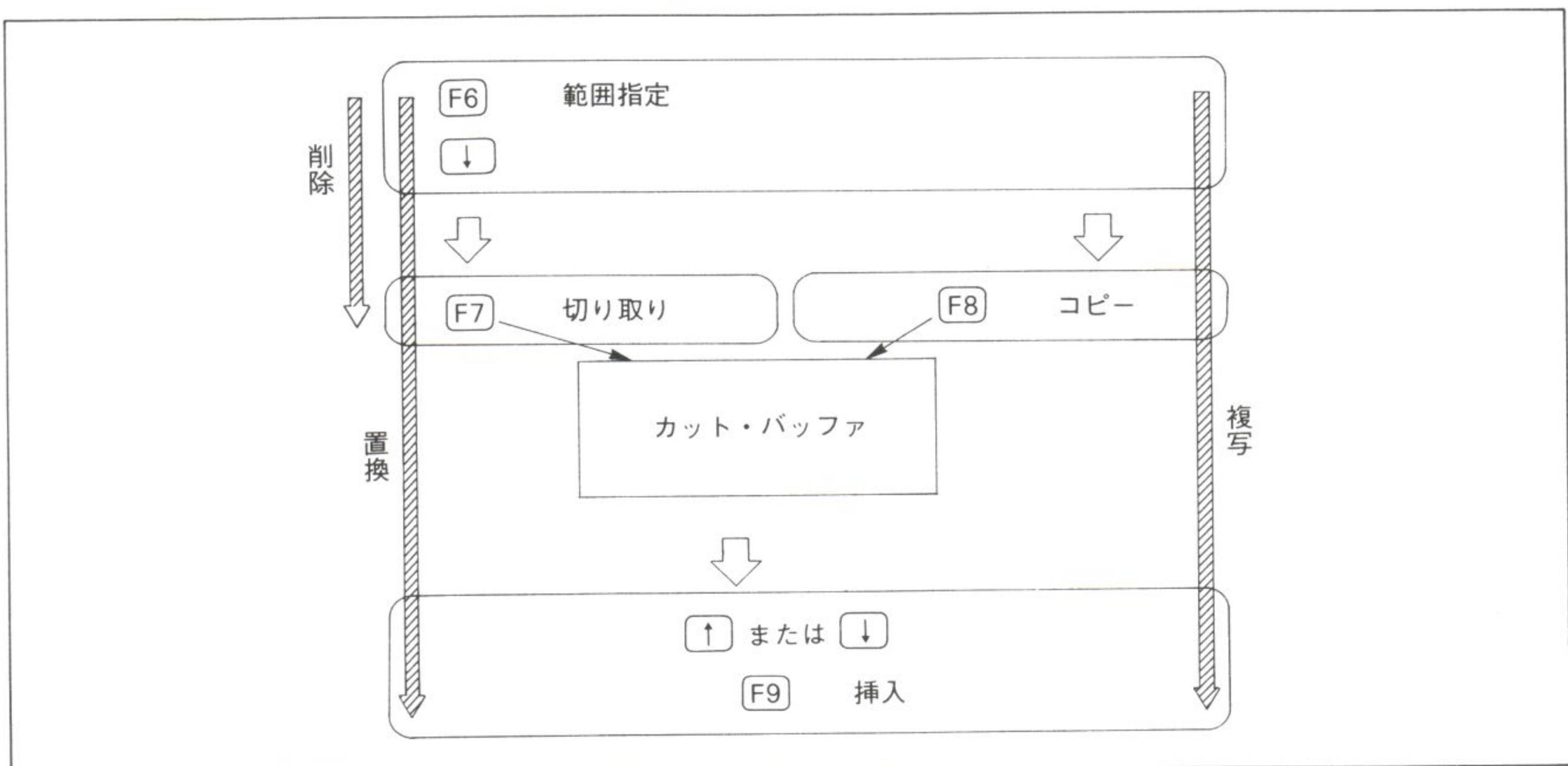
最もわかりやすい操作法は，図4.2のようにPFキーを使うことです．

F 6は範囲指定を開始するもので，続いて↓で希望する範囲の次の行までカーソルを移動させます．ここでF 7を押すと，指定された範囲の内容が消されてカット・バッファに入ります．その段階でストップすれば，テキストから削除されたままですが，↓または↑の任意の行にカーソルを移動後F 9を押すとカット・バッファの内容が該当テキスト位置に挿入されます．

カット・バッファに収容する際は，F 8を使うと，元のテキストから消されずにコピーされます．その後挿入手続き(F 9)をすれば，元のテキスト内容が複写されたことになります．

カット・バッファでは，収容されたデータは次のデータを収容するまで消えません．したがって，カット・バッファの内容は，何度でも挿入することができます．

●図4．2　カット・バッファを使った操作の関連

F6　範囲指定
↓
削除
F7　切り取り
F8　コピー
カット・バッファ
置換
複写
↑ または ↓
F9　挿入

4-4 ファイルの臨時入出力

サブルーチンの引用など，ソース・プログラム中に別なファイルの内容を挿入するといった処理が必要になることがあります．このようなときは，カーソルを挿入位置に合わせてから[ESC]，[Y]によって“ファイル読み込み：”メッセージを出させ，必要なファイル名を入力します．その結果，指定されたファイルの全内容が読み出され，カット位置から挿入されます．

こういった処理は，たいがい別ファイルの内容を末尾に付けるのが実態です．筆者などはエディタを立ち上げるのが面倒なので，このような場合，

type 〈別ファイル名〉＞＞〈主ファイル名〉

のようにすることがあります．こうすれば1回のコマンドでドッキングが完了し，別ファイルも残ります．続いて，その結果を確認するのならば，エディタを使ったほうがよさそうですが，テンプレート機能を使って，

type 〈主ファイル名〉

を実行することにより，内容の点検も簡単にできてしまうのです．ただし，このような「手抜き」は，大きなプログラムではしないほうがよいでしょう．なぜなら，大きなプログラムのときにはサブルーチンなどの挿入位置を，リストのドキュメント性を考慮した上で決めるべきだからです．

反対に，別ファイルに現在処理中のファイルの部分コピーを作りたいときは，[ESC]，[W]（“ファイル書き出し：”入力付き）を使います．このとき，事前のカーソル位置が重要で，[F 6]（範囲指定）を押すケースでは，押す前のカーソル位置が最初の行，押した後[ESC]，[W]直前の位置が最終行の1つ手前となります．また[F 6]を使わないときは，[ESC]，[W]の直前のカーソル位置が最初の行となり，ファイルの最後までコピーされます．

edでは最終行の扱いに注意!!

ed立ち上げ時に，－Lオプションを使って操作すると直接見えるのですが，各行の末尾には必ず[↵]が付いています．ただし，最後の行は[↵]なしで終わらせることができ，省略してもコンパイラやバッチ・ファイル動作には特別支障をきたしません．ところが，このようなファイルをtypeなどでプリンタに出力すると，最後の行が印字されないままストップしてしまいます．これは，[↵]コードがプリンタの印字指令になっているため，それ以前の行内容がプリンタ内のバッファまで届いていても印字できないからです．したがって，このようなときは

echo ＞prn[↵]

などで強制的にプリンタに[↵]コードを送ってやればなんとか切り抜けることはできます．

しかし，問題はそればかりではありません．たとえば[F 6]で範囲指定をする際，最後の行の次の行にカーソルを移動しようと思っても，[↵]がないと改行されないためうまくいきません．

たった“[↵]”ひとつのことですが，いろいろ問題を引き起こすので，あまり手抜きをしないほうがよいようです．

4-5 文字列の検索と置き換え

ソース・プログラムを修正するときには，エディタが立ち上がり，テキストがスクリーンに表示された直後，最初に修正する行にカーソルを移動させることから作業を開始するのが普通です．

簡単には，[F 4]を押して，

前方検索文字列：

の問い合わせに対し，該当文字列を入力すれば，現在のカーソル位置から下に向かって検索されます．また，逆方向(上)の検索は，[SHIFT]+[F 4]とします．同じ文字列を引き続き検索するときは，下方向には[F 5]，上方向には[SHIFT]+[F 5]を使います．

だんだん慣れてくると，文字列の検索と同時に修正する[F 3]を使うのが簡単でかつ強力です．このとき，

前方置換旧文字列：

と

前方置換新文字列：

の入力が要求され，それぞれの置換前，置換後の文字列を指定します．その結果，現在のカーソル位置から下の該当文字列がすべて置き換えられます．反対方向(上)には[SHIFT]+[F 3]を使います．検索中には，

検索中　ESC で中止

のメッセージが表示されるので，もし思ったより以上に置換しすぎているなどの異常に気付いて中止したいときは，[ESC]で止めます．

最も効率的に修正する方法は，テキストの始めから下の方向に順次訂正を進めていくことです．このためには，訂正事項が多いときはあらかじめ紙の上にリスティングし，赤ペンなどで訂正内容を書き入れておいて，それに従って訂正，挿入するのが一番よいでしょう．とくにプログラムの打ち込みや大きな修正の直後には，訂正箇所も多く，おおよその見当で作業をするとかえって二度手間になったりします．「いそがば回れ」と言ったところです．

4-6 複数ファイルの処理

エディタで複数ファイルを処理するには，立ち上げコマンド投入時に必要なファイル(10個まで)を並べて指定します．しかし起動した後は，今どのファイルを処理しているかを常に意識しておく必要があり，互いに関連のないファイルならばそれぞれ独立して起動させるほうが操作が簡単です．複数ファイル指定が効果を発揮するのは，他のファイルの内容の一部をコピーするとか，あるいは移動させるような場合です．

起動直後，スクリーンに表示されているのは最初に指定したファイルの内容です．他のファイルに切り換えたいときは，[ESC]，[A](アセンディング)により指定順，[ESC]，[D](ディセンディング)により指定の逆順に次のファイルに移ります．アセンディングでは最後のファイルの次は最初のファイルに戻り，ディセンディングでは最初のファイルの次は最後のファイルとなります．

ファイル内容の部分転送の手順は，4．3の「行の削除と移動，転写」のところで述べた方法で，1つのファイルの該当部分をカット・バッファに転送し，次に転送先にファイルに切り換えて挿入操作をします．

もし起動時に指定しなかったファイルを追加したいときは，[ESC]，[F]を押すと“編集ファイル：”と表示されるので，続いてファイル名を入力します．その結果，追加ファイル用の画面に切り換わります．こ

のようにして追加したファイルの順位は，起動時に指定したファイル群の末尾に置かれます．

複数ファイルを扱うとき最も注意しなければならないのは，ファイルのセーブ・タイミングです．処理ずみのファイルの内容を不注意によって誤変更しないためには，そのファイルのみに限定してセーブできます．その場合は，該当ファイルの画面に戻して[ESC]，[X]を押します．また，セーブしないで編集結果を無効にしたいときは[ESC]，[K]で終わらせます．これらの終結宣言によって，該当ファイルは処理順位の対象からはずされることになります．

残りのファイルを一度にセーブするときは，[ESC]，[E]が使えます．編集結果をセーブしないときに[ESC]，[Q]を入力するのも同じです．

4-7　単独キー系，CTRL系，ESC系コマンドの対応関係

以上の説明は，エディタの機能のうちエッセンスのみを紹介したものです．EDにはほかにもたくさんの

●表４．１　カーソルの移動操作一覧

カーソルの移動先	単独キー系	CTRL系	ESC系
スクリーンの先頭位置	[HOME]		
次ページ	[ROLL UP]	[CTRL]＋[C]	
前ページ	[ROLL DOWN]	[CTRL]＋[R]	
ファイルの先頭	[F･1]		[ESC]，[B]
ファイルの末尾	[F･2]		[ESC]，[Z]
改行(挿入時は行の分割)	[↵]	[CTRL]＋[M]	
上	[↑]	[CTRL]＋[E]	
下	[↓]	[CTRL]＋[X]	
左	[←]	[CTRL]＋[S]	
右	[→]	[CTRL]＋[D]	
右のタブ位置(右端では改行)	[TAB]	[CTRL]＋[I]	
指定行番号			[ESC]，[n]，[↵] (nは数字)
行の先頭(先頭にあるときは行末)		[CTRL]＋[B]	
行の左端		[CTRL]＋[Q]	
行の右端		[CTRL]＋[P]	
１ワード左		[CTRL]＋[A]	
１ワード右		[CTRL]＋[F]	

●表４．２　挿入操作一覧

挿入動作	単独キー系	CTRL系	ESC系
現在行の１行前		[CTRL]＋[N]	
現在行の次の行に挿入コピー	[F10]		
挿入モードのセット，解除	[INS]		

コマンドがあり，単独キー系，CTRL系，ESC系がそれぞれ入り混じっています．そこで，エディタの締めくくりとして，最後に個々の機能の各系列対比表(表4.1〜表4.8)を掲げ，読者諸氏の利用に供したいと思います．

●表4．3　削除操作一覧

削除動作	単独キー系	CTRL系	ESC系
1字削除	DEL	CTRL ＋ G	
1字前の文字を削除	BS	CTRL ＋ H	
1ワード削除		CTRL ＋ T	
現在行を削除		CTRL ＋ V	
CTRL ＋ Vの取り消し		CTRL ＋ L	
行先頭からカーソル前まで削除		CTRL ＋ U	
カーソルから行末まで削除		CTRL ＋ K	

●表4．4　検索，置換操作一覧

検索，置換動作	単独キー系	CTRL系	ESC系
下方向検索	F4		ESC ，N
上方向検索	SHIFT ＋ F4		ESC ，S
下方向検索続行	F5		
上方向検索続行	SHIFT ＋ F5		
下方向連続置換（確認付き）			ESC ，J
上方向連続置換（確認付き）			ESC ，L
下方向連続置換	F3		ESC ，R
上方向連続置換	SHIFT ＋ F3		ESC ，U

●表4．5　別ファイル操作一覧

別ファイル入出力動作	単独キー系	CTRL系	ESC系
別ファイル読み込み			ESC ，Y
別ファイル書き出し			ESC ，W

●表4．6　ペースト・バッファ操作一覧

カット・バッファ操作	単独キー系	CTRL系	ESC系
指定行数のバッファ格納			ESC ，n ，P ，↵
指定回数バッファから挿入			ESC ，n ，G ，↵
バッファ格納範囲指定	F6		
指定範囲をバッファ格納	F7		
指定範囲をバッファにコピー	F8		
バッファから挿入	F9		

●表４．７　ファイルの終了操作一覧

その他の編集動作	単独キー系	CTRL系	ESC系
ヘルプ・ファイル参照	HELP	CTRL＋J	
大文字←→小文字変換		CTRL＋I	
編集前の状態に戻す			ESC，O
ファイル名変更			ESC，T
Human 68Kコマンド実行			ESC，C
↵の表示（↓）および解除			ESC，M
大文字←→小文字変換モード切り換え			ESC，]
タブ記号の表示（→・・）			ESC，I
EOF*の表示および解除	CLR		

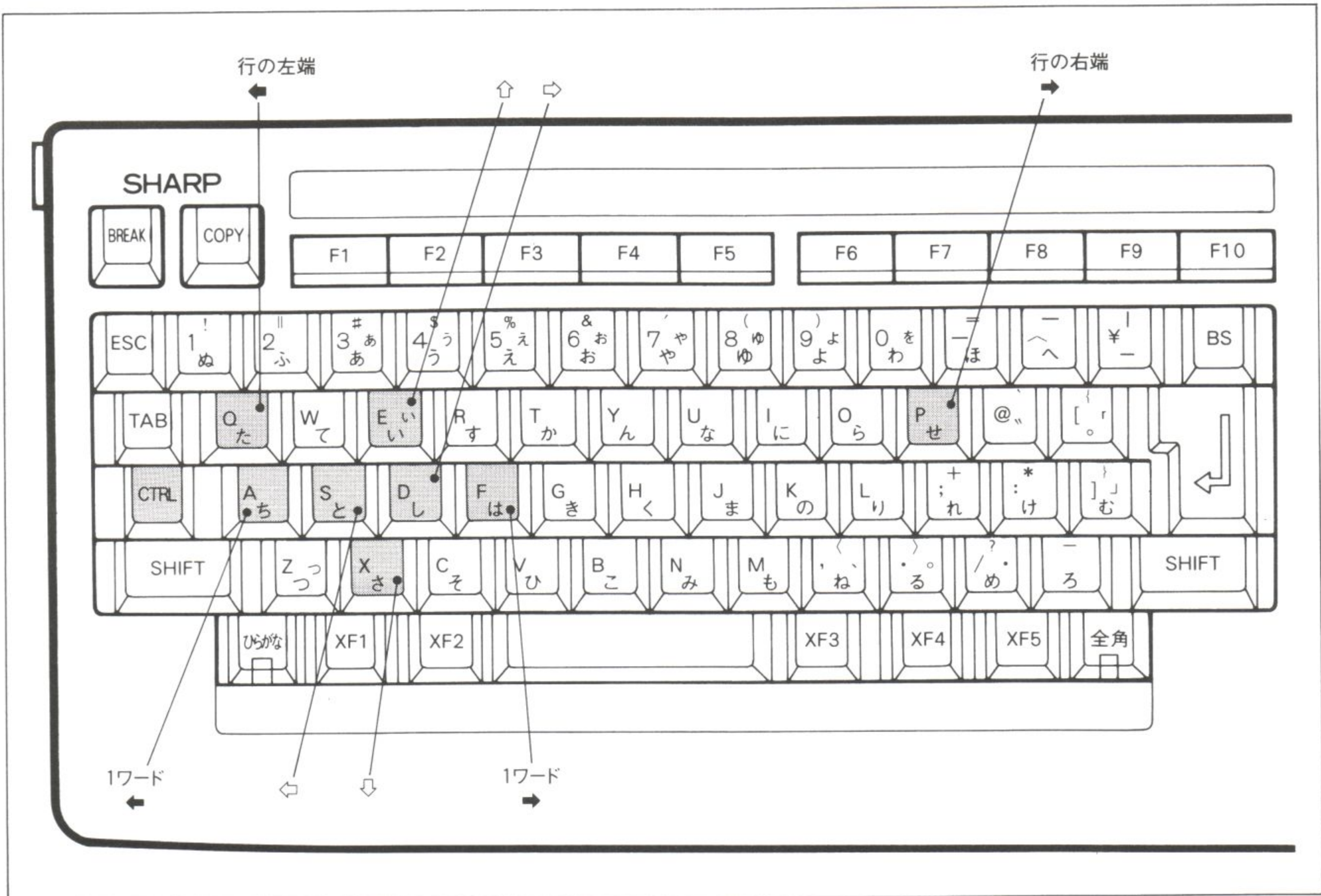
＊EOFはファイルの終端記号

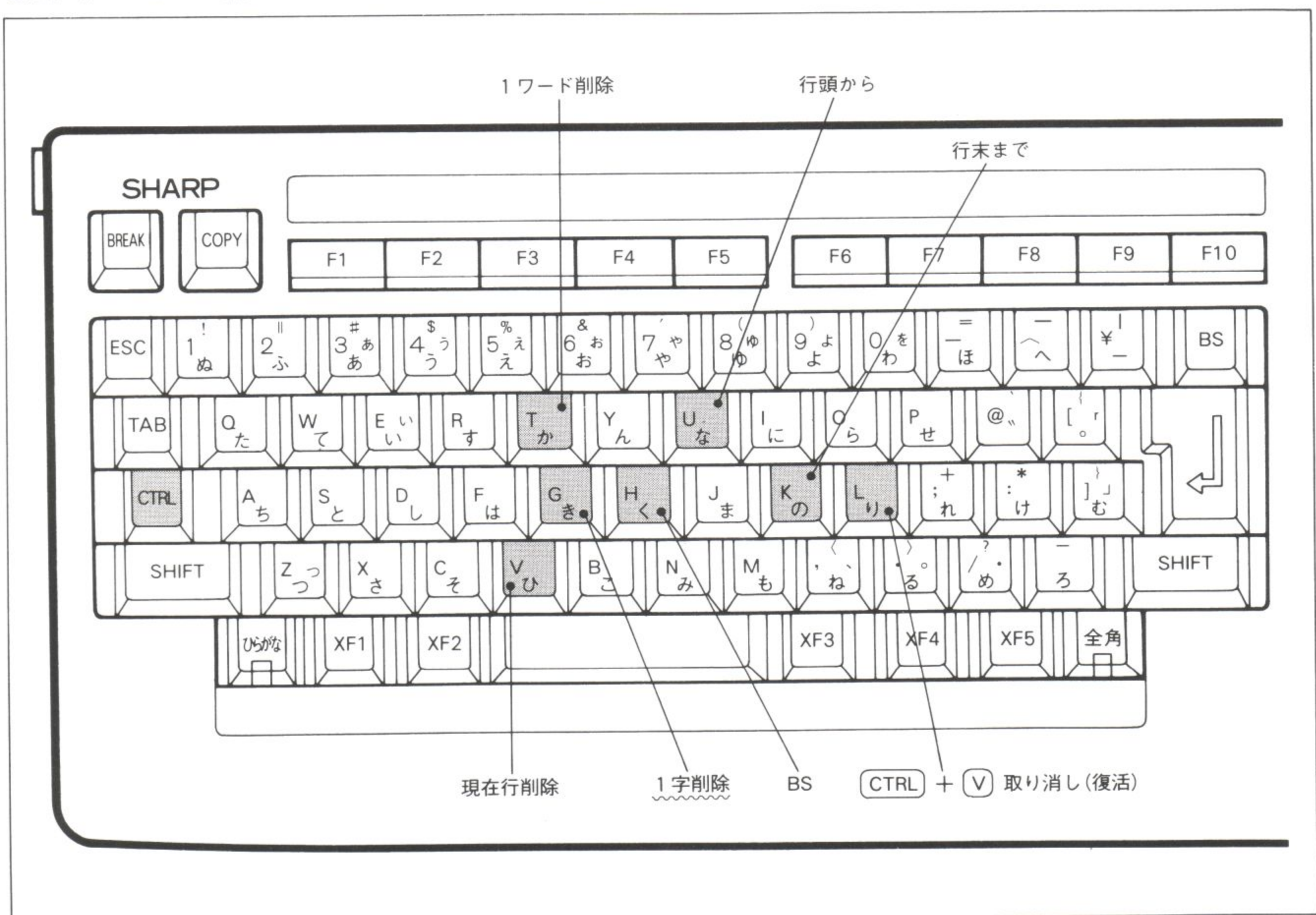
●表４．８　その他の操作一覧

ファイル・セーブ・終了・複数ファイルの動作	単独キー系	CTRL系	ESC系
編集中のファイルをセーブ			ESC，H
編集中のファイルをキャンセル			ESC，K
編集中のファイルをセーブし終了			ESC，X
起動したすべてのファイルをセーブし終了			ESC，E
起動したすべてのファイルをキャンセルし終了			ESC，Q
編集ファイルの切り換え（指定順）	SHIFT＋F6		ESC，A
編集ファイルの切り換え（逆順）	SHIFT＋F7		ESC，D
編集ファイルを追加する			ESC，F

また，カーソル移動に関するもの，削除に関するもののCTRL系コマンドは，まったくバラバラに考えられたのではなく，ある程度規則性をもっています．すなわち図4.3，図4.4のように，中心となるキーのまわりに，意味の似ているキーが隣接して配置されているので，このパターンを暗記すればいちいち表の内容を覚える必要がありません．

●図4．3　CTRL系コマンドのキーボードに対応した覚え方（カーソル移動）

行の左端 ⬅
⇧ ⇨
行の右端 ➡
1ワード ⬅
⇦ ⇩
1ワード ➡

●図4．4　CTRL系コマンドのキー分布（削除関係）

1ワード削除
行頭から
行末まで
現在行削除
1字削除
BS
CTRL ＋ V 取り消し（復活）

第3部

X-BASICプログラミング

第1章

X-BASICの考え方と基本的な命令コマンド

1-1 Cに似ているX-BASIC

X-BASICは,Cに変換できるよう意識して文法が決められているため,命令の中にはCによく似たものがたくさん含まれています．その中でもとくにその感じを強くするのは，変数の**宣言**です．マイクロソフト系など従来のBASICでは,データは宣言なしに引用ができました.このことは命令記述を軽減する反面,誤ったデータ名を書いても独立したデータとみなされ，ゼロなどの初期値が代入されたままプログラムが走ってしまうという問題がありました. X-BASICではこの点が厳密になり，変数を宣言するintなどの新しい命令が用意されています.

またCではサブルーチンは**関数**として定義，引用されますが，X-BASICでもgosub〜returnのほかに,関数名でサブルーチンを実行する機能をもたせています．このことによって，サブルーチンを独立させることができ，他のプログラムからの利用がしやすくなっています.

関数(サブルーチン)の定義は，funcに始まりendfuncで終わります．その中で，サブルーチン内だけで利用されるローカル変数の定義も可能です．関数ということは，実行後の結果の値を関数名で参照できることにもつながり,式の中で変数としても使えます.このほかにも,Cと同様カッコ内で引数を渡せるとか,**リカーシブ・コール***ができるなどといった高度な使い方ができるのがX-BASICの関数の強味です.

関数名によるサブルーチンの実行は，行番号の参照を不要にします.同じことは，if〜then〜elseなどの**制御構造**にも当てはまり，カッコを利用することによって複数の命令行を組み込めます．そうすることで,行番号をいっさい使わないプログラミングが可能になるのです.

コメントについても，X-BASICの書き方は,

＊リカーシブ・コールは「再帰呼び出し」と訳され，サブルーチンが自分自身をサブルーチンとして呼び出すことをいう.

X-BASICを使ってみた印象は，メーカーがBASICユーザーを抵抗なくCに誘導するために開発した言語ツールではないかということでした．実際，データ定義などはもちろんのこと，Cで使われている関数の一部までがBASICで使えるようになっています．意地の悪い見方をすれば，もともとCコンパイラを中心に開発し，BASICはそのオマケとして手数を省いて開発したと考えられないこともありませんが，ともあれその結果面白いものができてしまったのは事実です．

これを利用する側としては，C入門の予備ステップにもなり，またコンパイラにより実用的なプログラムも開発できるという特徴を大いに活かすべきでしょう．

このためこの章では，X-BASICの基本的な事柄を説明することにしたいと思います．

/＊ ～ ＊/

と，Cに習っています．

BASICソースをCに変換するプログラム（BASTOC）には，データ名の付け方など多少の制限はありますが，それらをクリアできればそれ以降はCのプログラムとして利用できます．このことは，コンパイルして最終的に**機械語ファイル**を作成すれば，BASICインタプリタ特有のスピードの遅さから開放されることを意味します．また，BASICインタプリタのままでは，プログラムを起動するとき**パラメータ**を渡せませんが，C変換すればそれが可能になります．

むしろ積極的に新しい命令体系を活用して，Cライクにプログラムを作っていくのがX-BASICの本当の楽しみ方です．

1-2 BASICインタプリタの起動と各モード

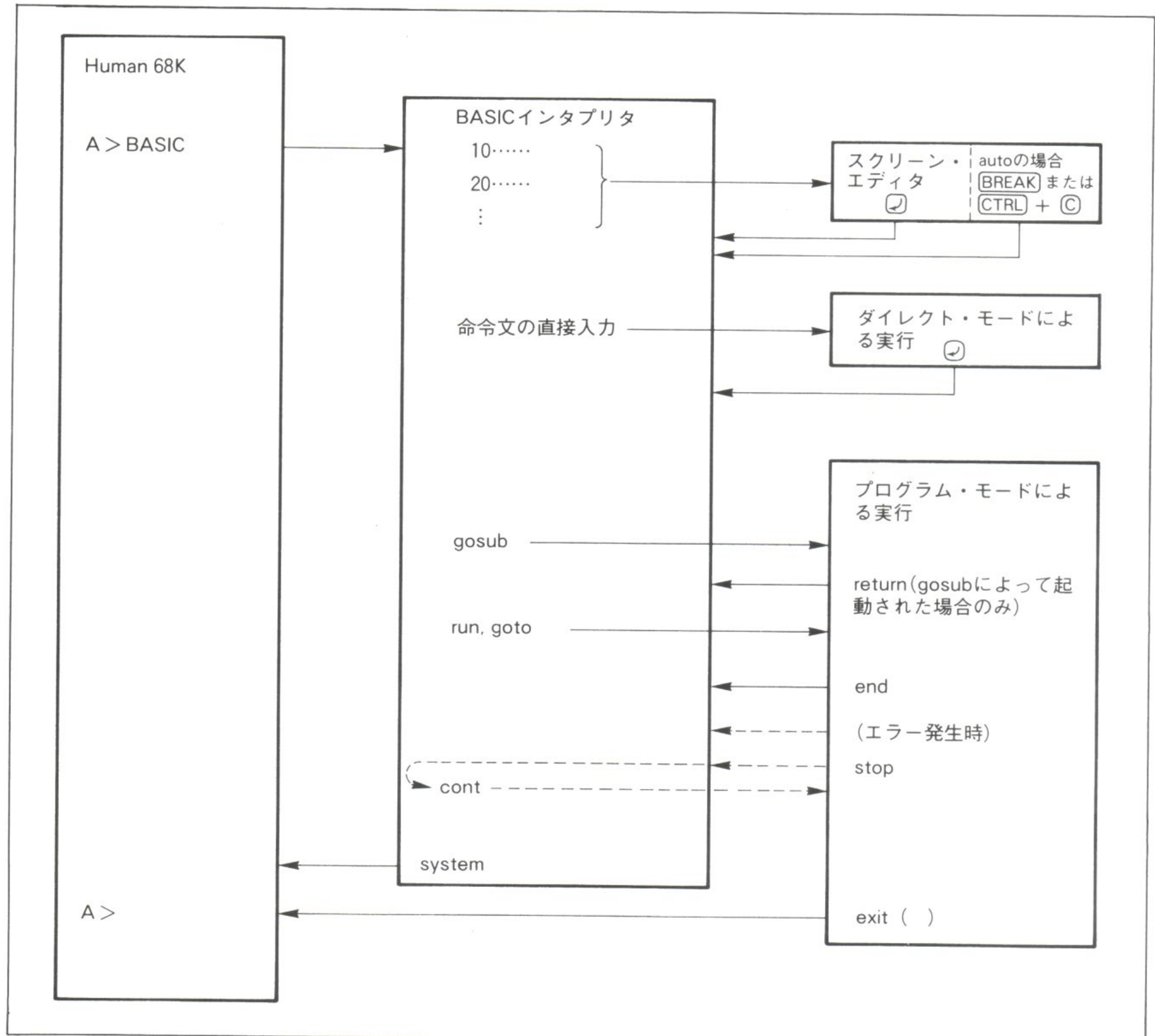
Human68KからBASICインタプリタを起動するには，

```
>BASIC
```

コマンドを実行します（図1.1）．起動後は，スクリーン・エディタ，ダイレクト・モード実行，プログラム・モード実行の3つの状態を往来でき，それぞれ図のようなコマンド関係で移行します．

スクリーン・エディタは，行番号からの入力で作動し，入力文字列の行末(↵)まで処理すると終了しま

●図1．1　BASIC インタプリタの各モードと移行の関連

Human 68K
A＞BASIC
A＞
BASICインタプリタ
10……
20……
⋮
命令文の直接入力
gosub
run, goto
cont
system
スクリーン・エディタ ↵
autoの場合 BREAK または CTRL ＋ C
ダイレクト・モードによる実行 ↵
プログラム・モードによる実行
return（gosubによって起動された場合のみ）
end
（エラー発生時）
stop
exit（　）

す．カーソルを以前の行まで戻して修正する場合は，その行で ↵ が押されないとエディタが働かないので注意が必要です．**auto** が指定されている場合は行番号が自動生成されるため，エディタの処理が連続します．この状態は，BREAK か CTRL ＋ C のいずれかを押すことによって解除できます．

行番号を書かずに命令文を直接入力すると，**ダイレクト・モード**で実行されます．この場合，↵ までが実行の範囲になるため，細切れに処理することになります．いわばこれは，BASIC の命令をコマンドとして利用するモードといえます．

行番号を付けて入力した命令ブロックは，通常 **run** 命令で実行されます．これが**プログラム・モード**です．このモードは **goto**，**gosub** 命令でも働かせることができますが，gosub については注意が必要です．サブルーチンの外部にあるそれは，int などの定義文が実行できないことです．反対に，プログラムの終了のために return 命令が使える（ただしサブルーチンがネスト*されているときは最も外側のもの）点は，この命令で起動するときのメリットといえるでしょう．

プログラム・モードはどの命令で起動しても，**end** 命令を実行すると解除され，インタプリタ本体に戻ります．**exit** 命令の場合は，BASIC インタプリタまで終了させ，ストレートに Human68K にバックします．いったんインタプリタに戻ってそこから Human68K に戻るには，**system** コマンドを使用します．

---

*ネストは「巣ごもり」の意味で，この場合サブルーチンの中にまたサブルーチンがあることを意味する．

1-3 スクリーン・エディタ関係のコマンド

❖関連コマンド　**new, auto, load, save, list delete, renum, files**

BASICのみならず，一般にプログラムはメモリに収容されて初めて実行可能になります．BASICプログラムを実行させるには，新規に入力するか，ファイルに収容されている既存のプログラムをメモリに展開しなければなりません．また，プログラムの修正にあたっても，テキストをメモリに置いてスクリーン表示を頼りに訂正，追加することになります．

新規にプログラムを入力する際は，もし何かのプログラムがメモリに残っているときは，newコマンドによって消去してから実行します．この場合，行番号がスクリーン・エディタで訂正する際の対応づけに利用されているので，増加方向に手動で入力するか，

auto [<最初の行番号>] [, <増分>]

コマンドで自動的に発生させます．このコマンドを使用すると，最初の行番号(省略値＝10)で指定された番号から，増分(省略値＝10)を加えて次々と新しい行ごとに番号が生成されます．この状態を解除するには，[BREAK]または[CTRL]+[C]を入力します．

また，既存のプログラムをディスクから読み込むには，

load "<プログラム名>"

コマンドを使います．プログラム名は，必要によりデバイス名や所属ディレクトリ名を先頭に付けることができます．このとき，以前のプログラムは消去されます．

ロードした内容は，

list [<開始行番号>] [ー [<終了行番号>]]

コマンドでスクリーンに表示できます．

スクリーン・エディタでの命令行の訂正は，訂正後のその行の内容を全部入力するか，スクリーンに表示されている該当行にカーソルを移し(⇦，⇨，⇧，⇩を使う)，修正した後[↵]を押すとその行の内容が書き換えられます．このとき[↵]を押さないと，その行は元のままになって残るので，スクリーンだけの書き換えで終わらないよう注意しなければなりません．

新しい行の追加は，追い番によって後に続ければよいのです．途中に挿入するときは該当位置の空いている番号値を使います．空き番を作る場合，またはプログラム完成後番号を整理するには，

renum [<新開始行番号>] [, <旧開始行番号>] [, <増分>]

コマンドによって再番号付けを行ないます．このコマンドは各行の先頭にある行番号部分だけしか処理せず，goto文などの行先きの記述まで訂正してくれません．このあたりがマイクロソフト系BASICに比べて弱いところです．しかし，この欠点も，行番号に依存しないプログラムを作ることによって回避できます．

プログラムの修正時の都合によって特定の行を消去したいときは，その行番号だけで命令のない行内容を入力すれば，行番号もろとも消えてしまいます．もし連続して複数行を消去したいときは，

delete [<開始行番号>] [ー] [<終了行番号>]

を使えば簡単です．

現在メモリにあるプログラムを，完成あるいは中途で保存する必要が生じた場合は，

save <ファイル名> [, <開始行番号>] [ー [<終了行番号>]]

でディスクに転送できます．このときセーブする範囲を行番号で指定できます．

X-BASICではBASICのもつスクリーン・エディタだけでなく，一般用のスクリーン・エディタ(ED)で作成したテキストも使えるようになっています．

EDによって手動で行番号を与えるときは，必ず昇順になるように注意しなければなりません．また，最終行を↵で終わらせないと読み込みの際エラーとなります．このことはスクリーンでは確認しずらいので注意が必要です．

ところでEDを使うと行番号の入力が大変なので，省略したいことがあります．もし省略したままのファイルをBASICでロードするときは，

load@　〈ファイル名〉 **[，〈開始行番号〉[，〈増分〉]]**

を使い，番号付けしてメモリに展開してやります．

反対に，行番号をはずしてセーブしたいときは，“save”の後に“@”を付けた形でコマンドを与えれば対応できます．行番号がなくなるとファイルの容量が節減できるので，このために省略することもあります．

load@コマンドでは，以前のプログラムは消去されません．したがって，開始行番号を，メモリに残っているプログラムとうまく調整することによって，複数プログラムの合併(マージ)が行なえます．

この方法を使えば，関数形式で作成しておいたサブルーチンをあちこちのプログラムから引用することができます．関数内で定義された変数は独立しているので，引用する側とデータ名が衝突していても心配はありません．

付け加えると，saveのときには，ファイル名が既存のものと同じであっても，警告なしに古いほうを削除してセーブしてしまいます．このため，事前に，

files　**[“〈デバイス名〉”] [“〈ファイル名〉”]**

でファイル・リストを表示させて確認するのが安全です．

1-4 データ型を定義する命令

❖関連コマンド　　**int, char, float, str**

マイクロソフト系など，一般のBASICはデータ型の宣言をする命令をもっていません．これらのBASICでは，データ型は，データ名の中に暗黙に示されています．この点でX-BASICは明示的に宣言できるところが長所となっています．

すなわち，データを定義する場所を明確にすることによって，それがメイン部(本体)ならば「**グローバル**

使えるkillコマンド!!

マニュアルには書いてありませんが，ファイルを消去するコマンドkillがX-BASICでも使えます．実験によれば，

kill　**“**〈ファイル名〉．**〈拡張子〉”**

で消せます．このとき拡張子まで指定しないとうまくいきません．BASICで拡張子を省略してセーブしたファイルは，“．bas”の拡張子が付いているので，もしこのコマンドを使うときは忘れずに付け加えるようにしましょう．

変数」となり，プログラム全体から参照できるという扱いを受けます．そして，関数(サブルーチン)内で定義されたデータは，「**ローカル変数**」としてその内部だけの参照にとどまります．言い換えると，ローカル変数は他の関数やメイン部とダブった名前を付けても衝突することはなく，好き勝手に命名できます．ただし，グローバル変数と同じデータ名を定義すると，その関数内から同名のグローバル変数を参照することはできなくなります．

データ型は，次の4種類のものがあります．

・int ………整数(4バイト)
・str ………文字例
・char ……整数(1バイト)
・float ……実数(8バイト)

定義するときは，データ型名から書き始め，続いて，“，”で区切ってその型に該当するデータ名を並べます．データ名の後に“＝”を付けて，それに続いて初期値も記述できます．

また，文字列はとくに明記しない限り32字までのメモリが用意されますが，“[”と“]”ではさんで，1〜255までのメモリ・サイズを指定できます．文字データを収容するときは，末尾に＄00(ストッパ)を付けて文字列の終わりとしているため，実際はさらに1バイト付加された状態になっています．

初期値データは数値の場合10進，8進(“&O”を先頭に付ける)，16進(“&H”を先頭に付ける)，2進(“&B”を先頭に付ける)のいずれかを選択できます．文字の場合は，“〜”の形で文字列が，また‘X’の形でchar型に代入できる文字の値が定義できます．char型は基本的には数値なので，文字コードに対応した数値に変換された状態で，データが保持されているように見えます．

1-5 配列の定義と参照

❖関連コマンド　dim

同名のデータを複数個定義するときは，配列を利用します．そしてこのための宣言には，

dim [<データ型>] <変数名> (<個数−1> {, <個数−1>}) [[<文字バッファサイズ>]]
[= {<代入定数値リスト>}]

を使います．

配列は基本的に同名項目の集まりなので，処理するときにはその中のどのデータを対象にするのかを明確にしなければなりません．たとえば同じaという名前のデータでも，aの3番と5番とではa(3)，a(5)のように区別して扱います．この場合，カッコ内の番号のことを「**添字**」と言います．X-BASICでは，添字は0から始まり，「**個数−1**」の項で指定された値まで使用できます．

X-BASICでは，配列に初期値を与えることができます．初期値データはたくさんの個数になることが多いので，“{”から書き始めると“}”が出現する行までの間に書けます．なお，個々のデータは“，”で区切って並べます．

dim文の書式は一見面倒そうですが，あとはデータ型を定義する命令と大差ありません．変わっているところといえば，配列の個数−1(添字の最大値)を追加定義しなければならないくらいです．

もともと配列は，メモリ上に同名データを並べただけの1次元のものですが，縦横の2方向に添字をもたせた2次元のものも想定できます．この場合，メモリ上では「何番目のブロックの中の何番目」といった取り扱いが行なわれます．同様な手法で3次元，4次元……といった配列が考えられます．いくら次元の数が大きくなっても，ブロックを階層構造にするだけなので，BASIC側ではメモリ上の対応アドレスを計算するのに困ることはありません．

このようにして定義された配列を参照するときは，同時に全部のデータを対象にすることはできません．

あくまでアドレスを特定できる1個のデータのみです．したがって，上述のa(5)のように変数名を添字付きで書きます．添字は定数で書くほかに，a(count)のように変数名で与えることもできます．

X-BASICは，マイクロソフト系のもののようにdata文で定義してread文で読み出すという機能をもっていませんが，data文に相当する機能はdim文の初期値設定を利用することで代替できます．1件ずつ取り出すのも，添字をプログラムで1ずつ増やしながら参照すればよいことです．

1-6 システム変数

❖関連コマンド　　date$, day$, time$, dskf, free

●日時を表わすもの-date$，day$，time$-

X-68000はカレンダー，タイマーを内蔵しています．これらの値は，参照したり，セットしたりできます．

日付のシステム変数は**date$**で，

yy／mm/dd

の形式により西暦年の下2桁，月2桁，日2桁の順の文字列となっています．

曜日を表わすのは**day$**です．この値は「日」～「土」の漢字(2バイト・コード)で直接表示されます．

時刻は**time$**で，

hh：mm：ss

形式で時2桁(24時制)，分2桁，秒2桁の順の文字列が与えられます．

これらの値は，式の中で右辺に書かれるなど，参照するときは現在値が読み出されますが，左辺に代入形式で書かれるときはセットすることを意味します．date$，day$は直接モードなどですぐにセットしても一般に困ることはありませんが，time$はキー入力している間にどんどん時刻が過ぎてしまうので，前もって先行した時刻を入力して，その時刻になったら，↵を押すようにすると正確にセットできます．

●システム資源の残量を示すもの-dskf, free-

BASIC操作中に知っておきたいことのひとつに，今現在どれだけのメモリが空いているかということがあります．たとえば，大きなプログラムでメモリがきつくなった場合などでは，このことはどれくらいの追加使用が許されるかという目安にもなります．また，子プロセスの同時実行の際にも，メモリ容量は最大の関心事に違いありません．

空きメモリの大きさは，**free**変数でわかります．

同様なことはディスクについてもいうことができ，大きなプログラムを入力したり，コンパイルしたりするときに使用する領域が充分かどうかということは常に知っておきたいものです．とりわけ大きなソースを入力するときは，ファイルにセーブする段階で満盃になったことを知ったのではお手上げです．それまで一生懸命キー入力したのが「水のアワ」となってしまいます．

ディスクの空きエリアの大きさは，

dskf(<ドライブ番号>)

変数によってわかります．ドライブ番号は，0がカレント・ドライブ，1がA，2がBに該当します．

これらの変数は，print命令によって表示させて参照するのが普通です．変数を左辺に置いて代入する(空きを増やすまたは減らす)のは無意味で，こういった使い方はできません．

1-7 実行制御のための命令およびコマンド

❖関連コマンド　　**run, goto, end, stop, cont, exit system, error on/off**

ここで取り上げる命令やコマンドは，BASICの各モードについての説明と重複するものもありますが，もっと詳細に掘り下げる意味で再度掲載しました．

BASICプログラムの実行は通常runコマンドによって開始されますが，このコマンドは，

run "**<プログラム名>**"

の形をとることができます．プログラム名を省略した形では，あらかじめメモリにロードされているプログラムを実行することが前提になっており，そうでないプログラムを実行するときのためにload+runの形態が用意されているのです．

さてプログラムの中では，普通は命令は行番号の小さい方から大きい方へと並んでいる順序に実行されていきますが，状況によってバックしたり，先の方に飛び越したりしたいことがあります．このようなときは，

goto **<行番号>**

命令を使い，その行番号へと続けることができます．ただしこの命令はプログラムの構造をわかりにくくし，リストを読みにくくする原因ともなるため，最近は嫌われる傾向にあります．X-BASICでも互換性のために残してありますが，むしろ使わないことを奨励しています．

たいがいのプログラムは終了とともにBASICのコマンド・モードに戻るよう作られ，このための終了命令としては，endが使われます．この命令はリストの終わりの位置にダミーとして書かれることもありますが，あくまで実行中のプログラムを終了させることだけが目的の命令です．

一方，一時的にBASICを立ち上げて実行する場合などは，プログラム終了と同時にBASICそのものを終わらせ，Human68Kの画面に戻したいことがあります．このようなときは，exit( )命令を使います．

また，デバック中には，プログラムの中途で止めてデータの途中結果を確認したいことがあります．stopはそのための命令で，contコマンドが投入されるまで実行を中断させる働きがあります．いったんコマンド・モードに戻ると，ダイレクト・モードによりprint命令でデータ内容をスクリーンに表示させたり，数式を使って変数値を変更することができます．このときlistコマンドも使えますが，プログラムの内容を変更するとcontで再開することができなくなります．なお，endで終了した直後も，print命令で最終のデータ値を参照できるので，不都合な結果で終わった場合などはこのことを活用するとよいでしょう．

プログラム内で外部関数を使用しているときは，その部分でエラーが発生するとメッセージを表示して終了します．もし無視して継続させたいときはerror off命令を前もって実行させておけばよく，エラー時終了の状態に戻すにはerror on命令を使います．

1-8 if~then~else~構文

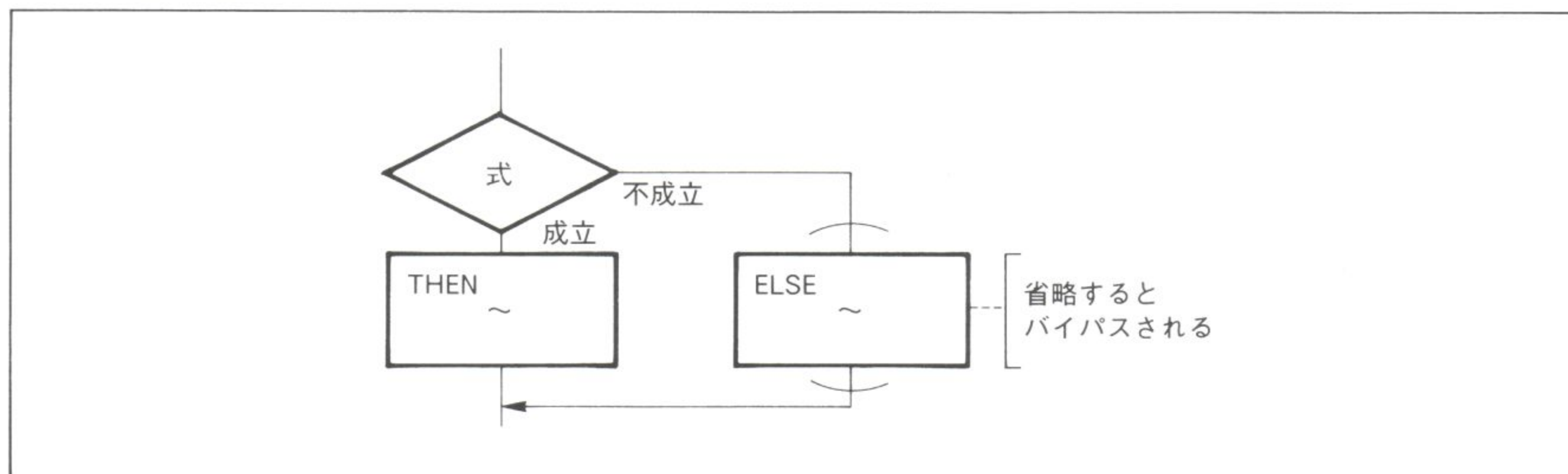
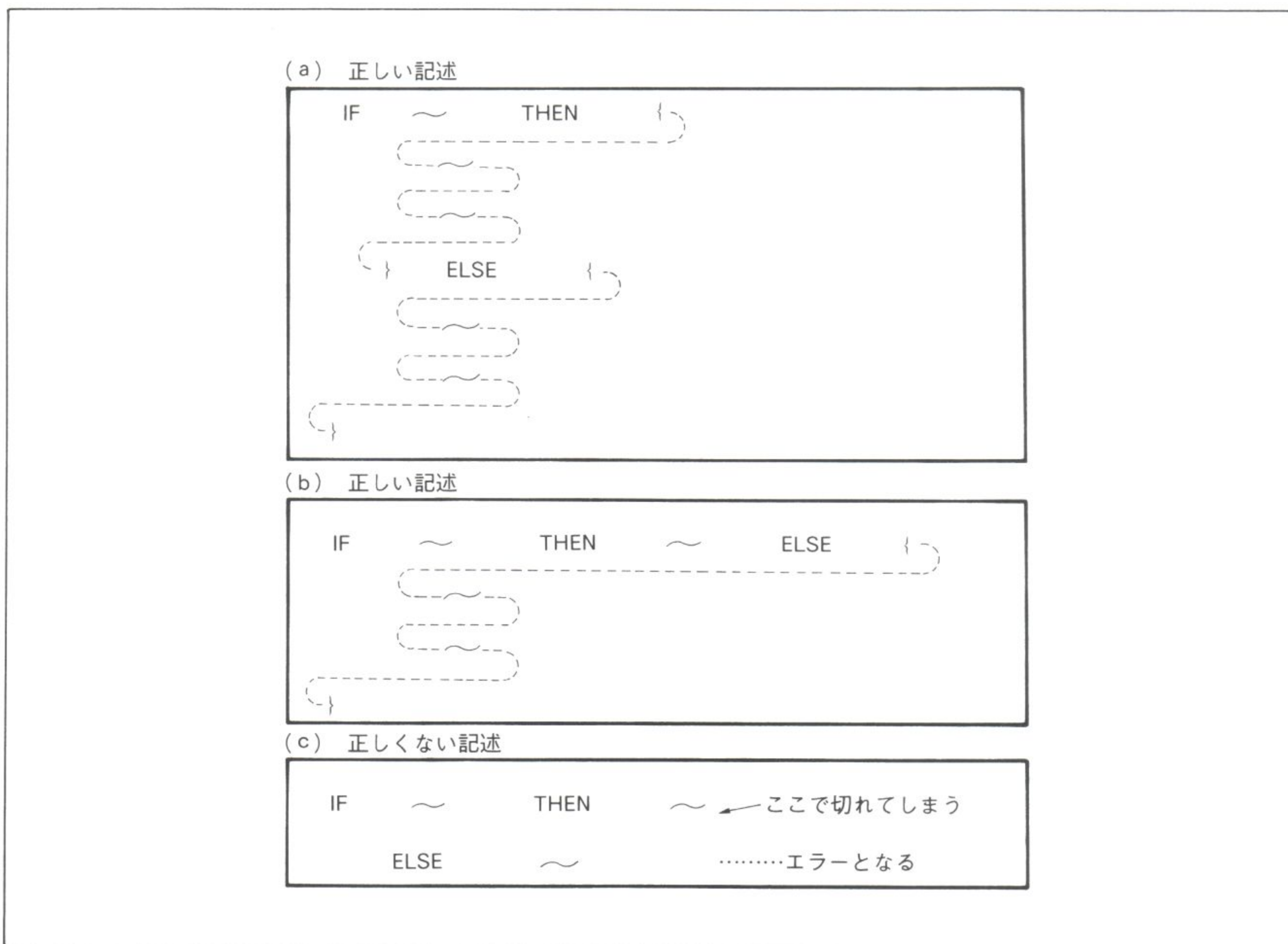
❖関連コマンド　　**if, then, else**

IF文は，図1. 2のフローチャートに描いてみるとわかるとおり，条件式が成立したときと不成立のときの処理を含む複合命令の形態になっています．このため，X-BASICでは，複数の行番号にまたがって記述できるよう配慮されています．ということは，またがらないで書けるようにもなっているので，どちらを

選択するのか明確に「意志表示」をしなければなりません．

図1.3の例では，(a)はごく一般的な記述の仕方で，比較的長くなる場合の命令の並べ方を描いたものです．このとき，文中に“{”が検出されると，以下“}”が出現するまでは記述が連続するとみなされ，図中の点線が示す区間において有効となります．このことは(b)の場合も同様で，点線部分はELSEの側の処理を説明しているものとして扱われます．

しかし，(c)の例では，THENの側の処理で終わっているとみなされ，次行のELSE以下がエラーとなります．これは，IF文がTHENの側の処理のみを記述することを許しているためです．見た目には(c)でも何ともないのですが，X-BASICでは，行をまたがるかどうかを“{”の相手（“}”）が未検出であるかどうかで判断しているので，こうなってしまうのです．もしELSE以下を次行に書きたいならば，(b)の形態で記述しなければなりません．

●図1．2　IF文のフローチャートとの対応

●図1．3　IF文を複数行番号にわたって書くときの考え方

1-9 for〜nextとwhile〜endwhile, repeat〜untilループ

❖関連コマンド　**for〜next, while, endwhile, repeat, until, continue**

同じ処理を繰り返すときには，X-BASICではfor〜nextなどのループを使い，goto文で最初に戻らせないようにするのがエレガントなプログラムを作るコツです．

for〜next, while〜endwhile, repeat〜untilの各ループは，図1.4のようにフローチャート化するとはっきりします．

for〜nextループは繰り返しカウントの初期値設定と増分(暗黙に1がとられる)加算後，終了条件(最大値)をオーバーしない限り処理が繰り返されます．すなわち，どんな条件のもとでも最低1回は処理が行なわれます．

ところがwhile〜endwhileのループでは，最初に条件判断をするようになっているので，始めから条件に合致していないときは1度も処理がなされません．また，条件式が不成立にならないと脱出できないので，処理の側で条件式の変数をコントロールしなければ，エンドレス・ループになってしまいます．このループでありがちなのは，以前の脱出条件のままで再度ループに入った結果，何も実行されないというパターンです．このようなときは，ループに入る直前に，式を成立させるための演算を行なわせなければなりません．

言い換えると，両者の違いは単に脱出条件を処理の後に調べるか，あるいは前に調べるかということだけではなくて，後者は繰り返しのための変数制御をまったく行なわないということに本質的な差があります．

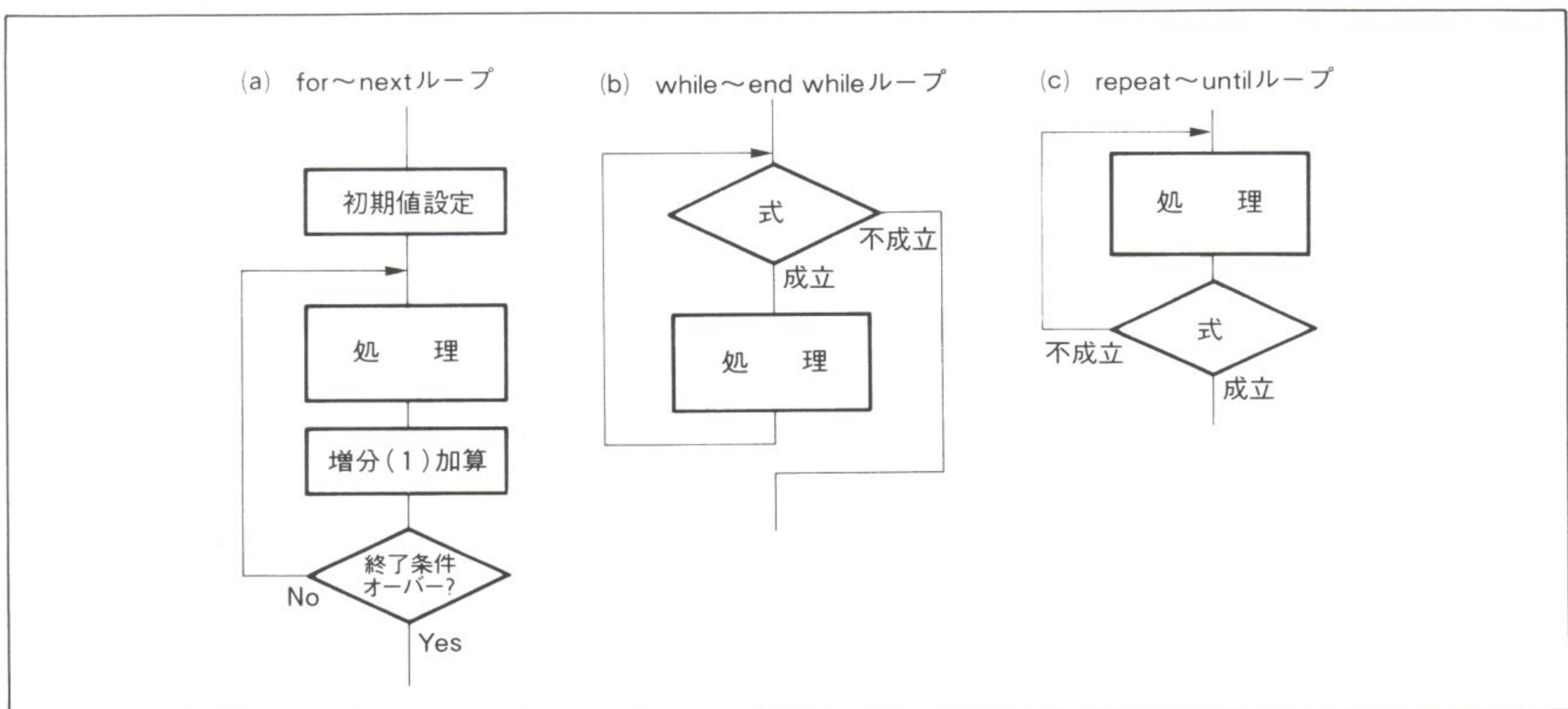
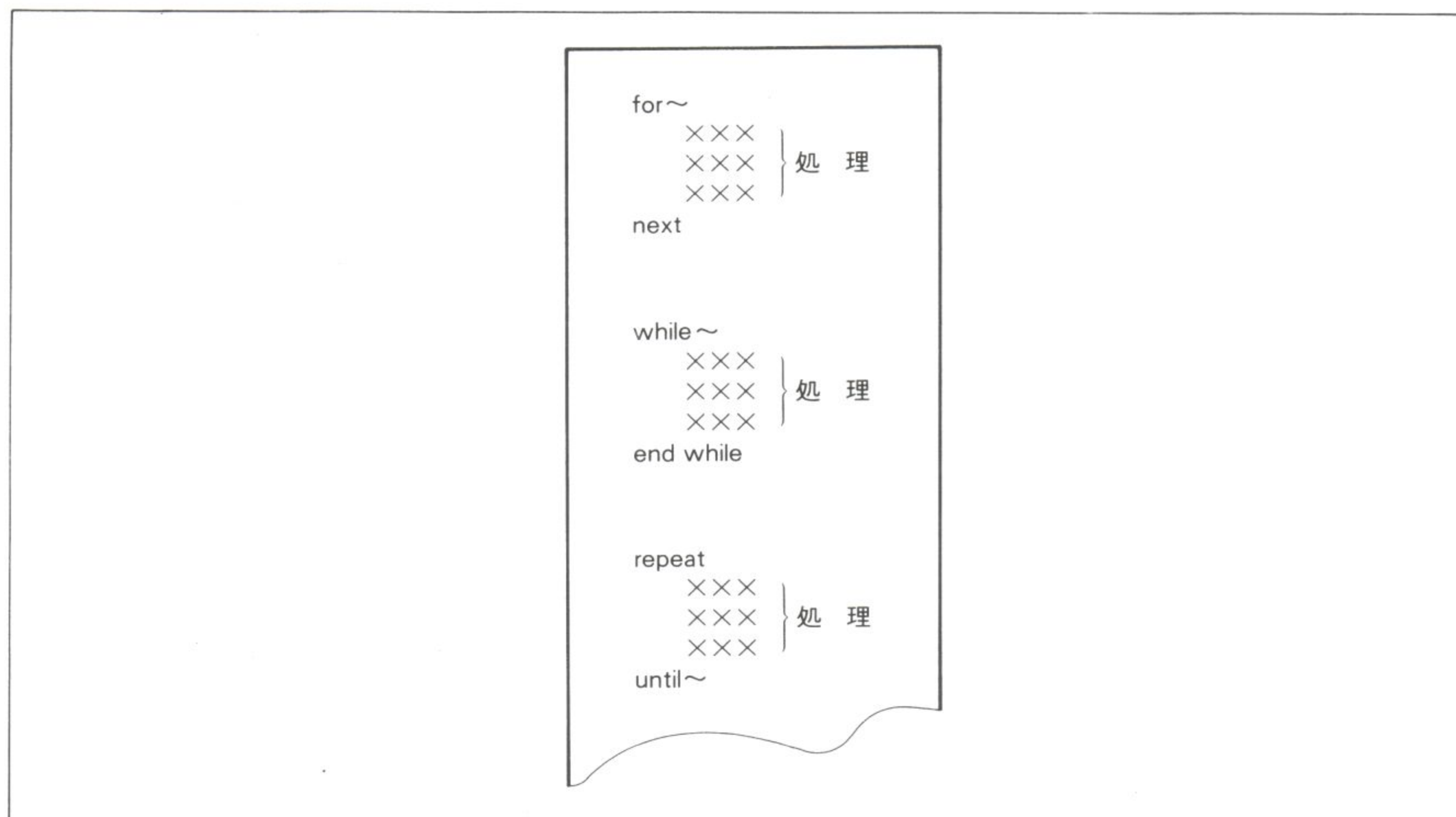
repeat〜untilも繰り返しのための変数制御をしないループの1つですが，脱出条件の判定が実行後なので，プログラム中1回は実行されます．

これらのループでは，プログラムを記述する際に図1.5のような字下げを行なって，ループ記述の始めと終わりを明確にすることがよく行なわれます．こうしておくと，繰り返される処理を他と区別することができ，プログラムの論理構造が把握しやすくなります．

これらのループの中で使われるcontinueは，いわばループの底(たとえばforならnext)へのGOTO文

●図1．4　X-BASICの3大ループ構造

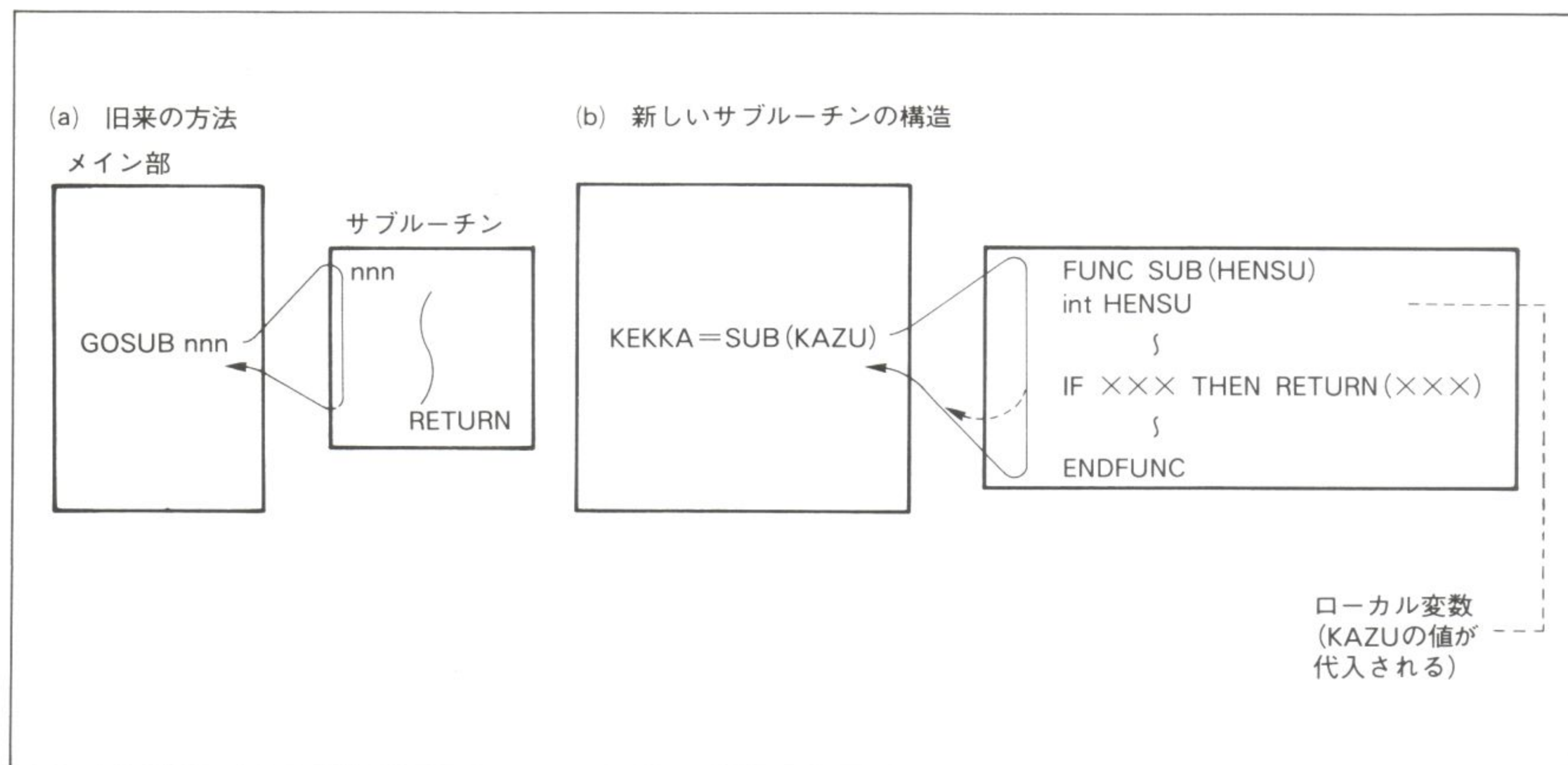
●図1．5　繰り返し処理の字下げ

```
for～
    ×××  ┐
    ×××  ├処　理
    ×××  ┘
next

while～
    ×××  ┐
    ×××  ├処　理
    ×××  ┘
end while

repeat
    ×××  ┐
    ×××  ├処　理
    ×××  ┘
until～
```

と考えるのがわかりやすいでしょう．

1-10 サブルーチンを作る新旧2つの方法

❖関連コマンド　**funk ～ endfunc，～ return**

従来の方法では，サブルーチンは普通の命令文で始まり，**return** 文で終わるように構成され，親となるルーチンからは **gosub** 文で呼び出されるようになっていました(図1.6(a))．しかし gosub 文は goto 文と同様に行先きを行番号で表わすため，参照されるサブルーチンの機能を想像することが難しく，またサブルーチンを他のプログラムで利用することも簡単でないという欠点がありました．

●図1．6　X-BASIC の新旧サブルーチン

X-BASIC では，こうした従来の方法も互換性の観点から引き継ぐ一方で，C 言語の関数のイメージを導入した新しいサブルーチンの形式を設け，奨励しています(図1.6(b))．

新しいサブルーチンの作り方は，

func [**<戻り値の形>**] **<関数>** (**<引数リスト>**)

に始まり，

endfunc

までの間に処理内容を記述します．

サブルーチン内で定義する変数は，「**ローカル変数**」と呼ばれ，そのサブルーチン内でのみ有効です．また，プログラムのメイン部(本体)で定義された変数(「**グローバル変数**」という)は，サブルーチン側で参照することも，変更することも自由です．

サブルーチンによっては，戻り値をもつものともたないものがあります．

戻り値をもつものについては，戻り値を設定するとき，

return (**<戻り値>**)

命令を使います．この命令はサブルーチンを終了させる働きをもっているため，戻り値をもたないやり方で，

return ( )　　　(戻り値省略)

のように使用することもあります．

サブルーチンの参照は，たとえば“sub”というサブルーチン名の場合，

sub (**<引数リスト>**)……………………戻り値なし
<戻り値>＝sub(**<引数リスト>**) ……戻り値あり

のように関数名を直接記述します．いわば，新しい命令ができたのと同じことになります．

1-11 多重分岐構文

❖関連コマンド　　switch ～ case ～ default ～endswitch, break

条件分岐が重複するときは，多重分岐構文を使うとプログラムが見やすくなります．

この構文の書き方は，図1.7のように switch～endswitch の間に，case による各条件ごとの処理文を並べます．default は，各条件が式０に該当しなかったときの実行文ｄのために最後に置くもので，必要がないときは省略できます．

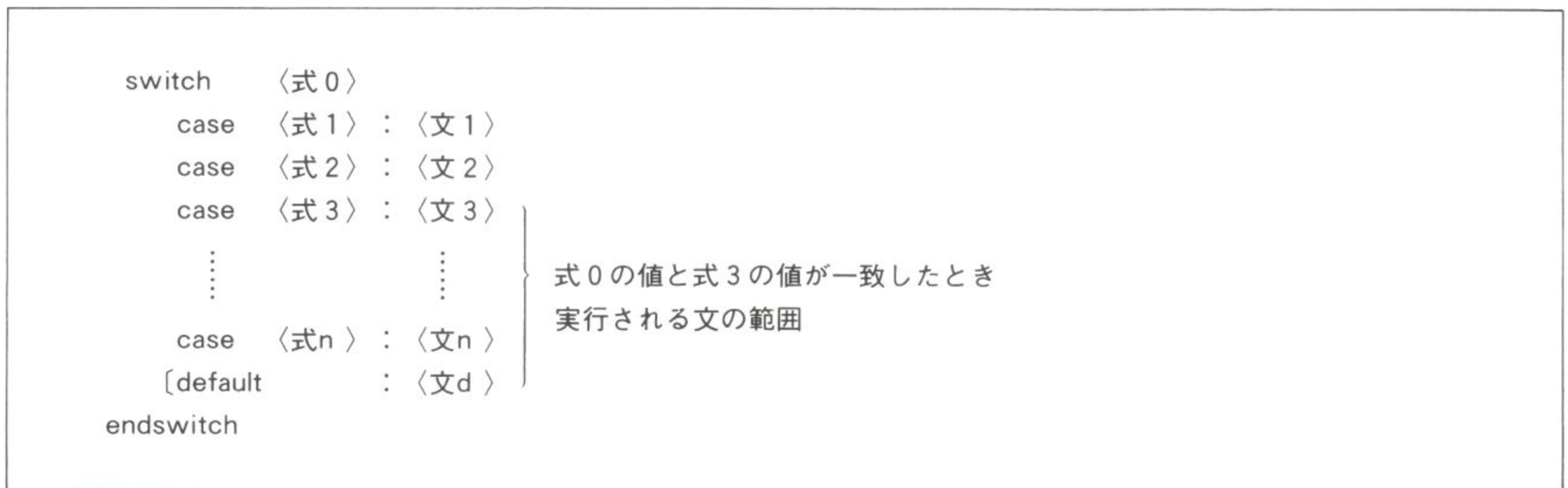
式は，たとえば変数ａの値によって分岐するとき，

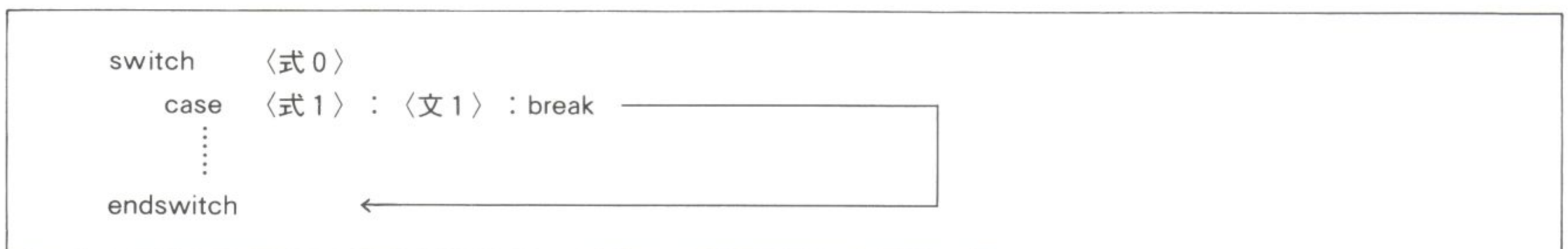
a ……式０
0 ……式１(a＝０ならば文１以下実行)
1 ……式２(a＝１ならば文２以下実行)
　⋮

のように書きます．このとき，case の内容は上から順番に検査され，条件が一致した行と以下の行の文の内容が有効になることに注意が必要です．すなわち式１で一致した場合は，全部の文の内容が上から順番に実行されるのです．

このようなことを防ぎ，一致した行だけの文を実行させるには，図1.8のように文末に break 命令を入れます．これによって，１つの文の内容が実行し終わると endswitch の位置に抜けるようになります．

●図１．７　多重分岐構造で実行される範囲

```
 switch   〈式0〉
    case  〈式1〉：〈文1〉
    case  〈式2〉：〈文2〉
    case  〈式3〉：〈文3〉 ┐
      ⋮            ⋮      │ 式0の値と式3の値が一致したとき
                          │ 実行される文の範囲
    case  〈式n 〉：〈文n 〉│
   〔default     ：〈文d 〉┘
endswitch
```

●図１．８　多重分岐構文で不必要な文をバイパスする方法

```
switch   〈式0〉
   case  〈式1〉：〈文1〉：break ─────┐
     ⋮                                │
endswitch       ←─────────────────────┘
```

1-12 Human68K のコマンド実行

❖関連コマンド　!

X-BASIC では，BASIC から Human68K のコマンドを実行できます．そして，このための記述形式も，

!　<Human68K コマンド>

と簡単です．

ただし，このときコマンド・プロセッサ(command. x)が存在していなければならず，かつそれをロードできる空きエリアが BASIC のテキスト・エリアとは別に用意されていなければなりません．

実験では，カレント・ディレクトリと同一ディレクトリ上に command. x を置かなくても，ルート・ディレクトリ直属の状態(メーカー出荷時のまま)で，任意のディレクトリから実行できました．

なお，実行できるコマンドは内部コマンドで，command. x だけで実行できる範囲です．

第2章

数値や文字列の処理

2-1 一般関数

❖関連コマンド　　abs, fix, int, sgn, sqr, log, exp, pow, pi

BASIC の関数の中には，符号や端数を処理するグループがあります．

abs は絶対値をとるためのもので，負数は正数に変換されます．反対に符号だけをとるのは，sgn です．この関数は正なら1，負なら－1を戻り値として設定します．

●図2．1　ABS，FIX，INT，SGN のテスト・プログラムと実行結果

```
  10 float a,b
  20 a=2.1#
  30 b=-2.1#
  40 print "fnc"," A"," B"
  50 print " = ",a,b
  60 print "ABS",abs(a),abs(b)
  70 print "FIX",fix(a),fix(b)
  80 print "INT",int(a),int(b)
  90 print "SGN",sgn(a),sgn(b)
 100 end


fnc        A        B
 =         2.1     -2.1
ABS        2.1      2.1
FIX        2       -2
INT        2       -3
SGN        1       -1
```

数値や文字列の処理は，演算式や関数によって行なわれます．このため，基本的な命令では他の BASIC と大差ないものの，C で使われている関数がたくさん顔を出すなど，かなり新しい要素が加わっています．

しかし，本章ではとくに新旧の区別をせず，あくまで機能中心に考えた関数の使い方について説明します．似たような機能のものが多くなっているので，ここではそれぞれの関数の違いを明確にし，どれを使ったらよいかという指針にもなるように工夫したつもりです．

端数処理を行なうものの１つは fix です．この関数は小数点以下を切り捨てる働きがあります．似たような動作を行なうものに int がありますが，この関数の定義は「変換前の値を越えない整数」を戻り値とすることになっているので，負数の扱いが fix と異なります．

これらの関数を比較するために作成した BASIC プログラムと，その実行結果を図2.1に示します．参考までに付け加えると，float 型から int 型にデータを移すことによっても小数点以下を切り捨てることができます．このことは，プログラム・ミスの原因になりがちなので注意したいものです．

このほかに比較的よく使われるものとしては，平方根を計算する sqr，自然対数を求める log，これと逆関数の関係にある exp などがあります．他の BASIC と同様，X-BASIC では複素数はサポートされていないので，負数の平方根は計算できないことに注意が必要です．平方根と反対の関係にあるべき乗は pow 関数を使います．この関数は，第１パラメータの値を第２パラメータ乗するものです．

X-BASIC では，π（＝3.14……）の値を関数 pi としてもっており，π の値とパラメータ値との積を戻り値として得ることができます．たとえば，2π は“pi(2)”と書けばよいのです．

2-2 三角関数と乱数を扱う関数

❖関連コマンド　**sin, cos, tan, atan**
srand, randomize, rnd, rand

X-BASICではsin，cos，tan，atanの三角関数を用意しています．関数名からもわかるように，これらはサイン，コサイン，タンジェント，アーク・タンジェントに対応しています．

三角関数の演算に際しては，度数法（1周を360°として計算する方法）と孤度法（1周を2πラジアンとして計算する方法）とがありますが，X-BASICでは**孤度法**を採用しています．もし度数法で計算したいときは，

$$ラジアン値=2\pi\times\frac{度数値}{360}$$

によってラジアン値を求め，この値を使って三角関数を求めます．

またX-BASICでは，乱数を扱う際に2つの系統を選択できます．

1つは，rnd( )の系列で，0以上1未満の範囲のfloat型で乱数を発生します．もう1つはrand( )系列で，0以上32,767以内のint型で乱数を発生します．

どちらの乱数系列もソフトウェアで発生させる以上，何もしないと常に同じ数から始まり，2番目，3番目……の各値が毎回同じように続きます．これでは乱数の意味がなくなるので，毎回異なった数から始まるよう「タネ」を用意します．

rnd( )に対しては，randomizeが，rand( )に対してはsrand関数がその働きをするもので，これらの関数にそれぞれ－32,768～32,767，0～65,535の範囲で適当な数値を与えると乱数系列を初期化できます．これらの値もプログラムで固定値にしてしまうと無意味なので，日付，時刻（システム変数として参照可）などから合成することがよく行なわれます．

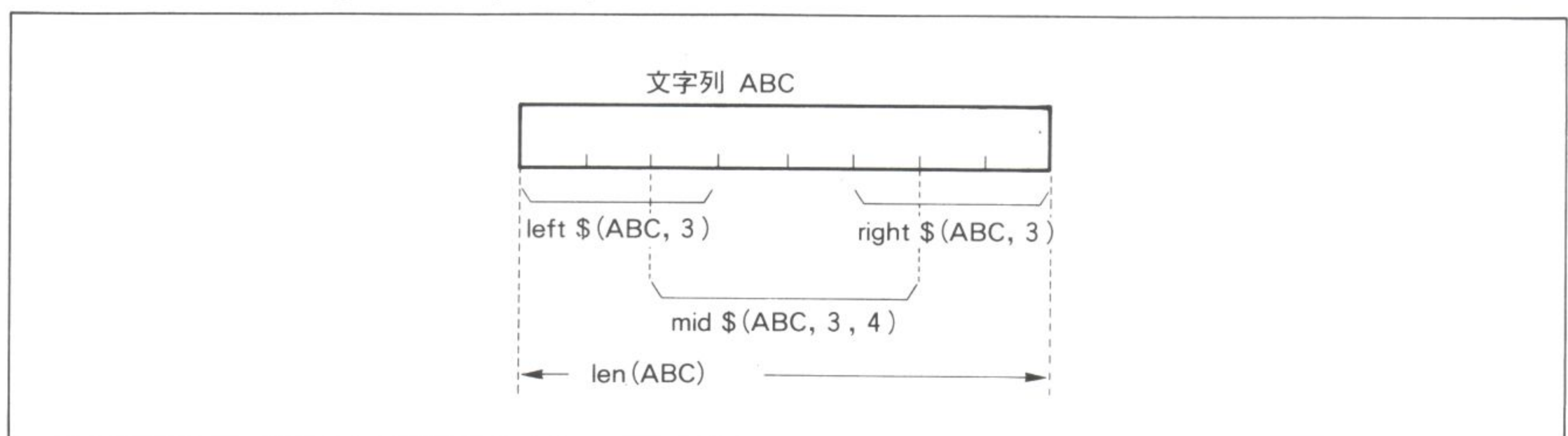
2-3 文字列を操作する関数

❖関連コマンド　**rights, mid$, left$, len, mirror$,**
strrev, strlwer, strupper

文字列は，任意の長さに切ったり，つないだりできます．

つなぐ場合は，

cstr＝astr＋bstr

●図2．2　文字列とright$，mid$，left$，lenの関係

のように記述すると，式の順序に接続されたcstrができます．

切る場合は，文字列をどのように切るかによって**right$**（右から），**mid$**（中間），**left$**（左から）のいずれかを選択します(図2.2)．各関数は第１パラメータが文字列名で，right$，left$では第２パラメータが切断後の文字数です．mid$では，第２パラメータが切断開始位置，第３パラメータが文字数となります．たとえば，

```
xyz=mid$(abc, 3, 4)
```

と書くと，文字列abcの３字目から４字分のデータを切り取り，文字列xyzに代入することを意味します．このとき，元のabcの内容は変化しません．元の値も変化させたいときは，左辺の文字列名を右辺と同じにしてください．

なお，文字列の長さを知りたいときは，**len**関数で戻り値として取得できます．len関数は，X-BASICではstrlen関数と同じです．命令は短いほど誤記を防げるので，前者を使うようにしたほうがよいでしょう．

文字列の内容を逆順にしたときは，**mirror$**か**strrev**関数を使います．いずれも戻り値として前後が反転した結果を得ることができますが，strrev関数は元の文字列まで反転してしまうので，戻り値のない形式での記述もできます．

文字列の内容は大文字だけ，あるいは小文字だけにしておくと，検索などのときに好都合です．なぜなら，同じ文字でも大文字と小文字ではコードが異なり，CPUは別な文字として認識してしまうからです．小文字だけの文字列に変換するときは**strlwr**，反対に大文字だけの文字列にするときは**strupr**関数を使います．いずれも元の文字列を変更するので，戻り値を省略できます．

2-4 文字，文字列を検索する関数

❖関連コマンド　**instr，strchr，strrchr　strspn，strcspn，strtok**

文字列の中から特定の文字の並びを検索したいときは，**instr**関数を使います．図2.3はその使用例で，パラメータに検索開始位置，文字列名，文字の並びの順で与えます．このときの戻り値は，見つかった文字の並びの先頭位置を指しています．

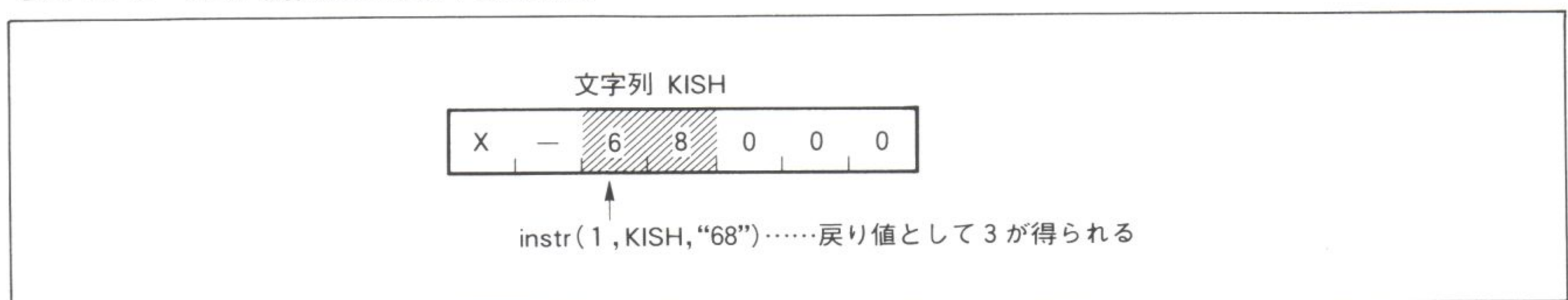
文字列中から特定の文字を検索するときは，instr関数で１文字だけの文字の並びを指定すればよいのですが，**strchr**関数を使って検索することもできます．この関数は，第１パラメータが対象文字列，第２パラメータがchar型定数となっているので，検索したい文字値は数値に変換して与えます．言い換えると，文字コードで検索するのに適した関数であるといえます．この場合注意しなければならないのは，戻り値が“見つかった位置－１”になっていることです．

検索文字を複数の「候補」で指定するには，**strcspn**関数を使います．この関数では検索文字(第２パラメータ)を候補文字を並べた文字列の形で与え，そのうちのいずれかの“文字が見つかった位置－１”が戻り値として返されます．また，見つからなかったときは，文字列の長さが戻り値となります．対象文字列が空文字ならば，検索は行なわれず，したがって戻り値も０のままです．

●図２．３　instr関数による文字列の検索

文字列 KISH

X	－	6	8	0	0	0

instr(1, KISH, “68”)……戻り値として３が得られる

反対に，検索対象外文字(複数も可)を与えて，それら以外の文字を検索するのがstrspn関数です．第2パラメータの戻り値では，対象外文字群を並べて文字列とする以外は，strcspnと同じです．

以上の関数は最初に見つかった段階で検索を終了しますが，同じ文字を最後まで繰り返し検索するのがstrrcharです．これはstrcharに対応する関数で，文字値はchar型定数で指定します．

一方，候補文字群のいずれかが見つかるまでの文字列を抽出するためには，strtok関数が用いられます．この関数は，“，”などで区切られた記述事項を対象文字列から1件ずつ取り出すのに好都合で，第2パラメータは候補文字群の文字列を書きます．同一の対象文字列を継続して扱うには，2回目以降の第1パラメータを空文字にしておくと，1回目の第1パラメータで指定された対象文字列の現在点(前回候補文字が見つかった次の文字位置)から継続して処理されます．

2-5　文字を検査する関数と変換する関数

❖関連コマンド　**is □□□□, toascii, tollover, toupper**

文字列の処理に際しては，個々の文字がどのような種類のものであるかを調べることも多く，そのような場合は表2.1に示す関数を使います．これらにおいては，検査の結果該当したときは－1，そうでないときは0が戻り値として得られます．

たとえば，入力された文字が数字かどうか調べたいときは，isdigit関数を使います．このとき検査の結果数字を表わす文字，たとえば3ならば，戻り値は－1となります．

文字の検査に際しては，コードそのものを変換してしまいたいことも少なくありませんが，以下に述べる各関数はそういった用途に使えるものです．

toasciiは，ASCII文字(&H00～&H7F)以外の文字コードだった場合，ASCII文字に変換する関数です．ASCII文字の特徴は最上位ビット(D_7)が0であることで，変換するといっても単に最上位ビットをクリアする程度のものです．そもそも，カナ文字をアルファベットに意味のある変換はできないのです．

tolowerは，英大文字のときに英小文字に変換します．また，toupperは反対に英小文字だったら大文字に変換します．

これらの変換用関数は，変換対象文字が変換条件に合致しないときは，そのままの値を戻り値として返します．

●表2．1　文字値を検査する命令と戻り値＝－1となる応答条件

関　数	戻り値＝－1にする文字の種類	具体的な範囲
ISALNUM	アルファ・ニューメリック(英数字)	“a”～“Z”，“A”～“Z”，“0”～“9”
ISALPHA	アルファベット(英字)	“a”～“z”，“A”～“Z”
ISASCII	ASCII文字	&H00～&H7F
ISCNTRL	CTRL (コントロール)文字	&H00～&H1F，&H7F
ISDIGIT	数字	“0”～“9”
ISGRAPH	表示可能な文字(スペース除く)	&H21～&H7E
ISLOWER	英小文字	“a”～“z”
ISPRINT	表示可能な文字(スペース含む)	&H20～&H7E
ISPUNCT	表示可能な記号文字	&H20～&H2F，&H3A～&H40， &H5B～&H60，&H7B～&H7E
ISSPACE	表示するとスペースになる文字	
ISUPPER	英大文字	“A”～“Z”
ISXDIGIT	16進文字	“0”～“9”，“a”～“f”，“A”～“F”

2-6 特定の文字を文字列に埋める関数

❖関連コマンド　**space$, string$, strset, strnset**

スペースだけの文字列を作るなど，文字列の中を特定の文字コードで満たしたいことはよくあるものです．

最初に紹介する **space$** 関数は，続く( )内のパラメータで指定された長さのスペース(&H20)文字列を作るのに使います．

スペース以外の文字に対しては，

　　string$ (<字数>，<文字コード>)

関数で，文字のコード値または文字そのものをオーダーしてやります．なお，文字を直接与えるときは，引用符(")で囲みます．

以上の関数は，始めから特定文字で満たされた文字列を作るのに使われますが，現在何らかの文字が入っている文字列を，別な特定文字で埋め直したいときは，別な関数によらなければなりません．

strset 関数は，第１パラメータで対象文字列を指定し，第２パラメータで満たしたい文字コード値を与えると，もとの内容が消えてすべて新しい文字コードに置き換わります．

文字列全部でなく，先頭から任意のバイト数だけにとどめたいときは，

　　strnset (<対象文字列>，<文字コード値>，<文字数>)

を使います．この関数では，指定された文字数に従って文字列の先頭から埋めていくため，文字数が１より小さい(０または負)場合は何もしないで終了します．

以上の関数のうち space$，string$ は，引用符で囲んで文字列で代替できるため，数文字程度で定義する場合はあまり使用するメリットがありません．文字列が長くなって字数を数えるのが大変な場合とか，字数が可変の場合にその便利さが光ってきます．

strset, strnset は，戻り値のいらない関数で，結果は第１パラメータで指定された文字列に直接入れられます．

2-7 異なったデータ型に変換する関数

❖関連コマンド

asc, atoi, atof, itoa, val
chr$, str$, bin$, oct$, hex$

X-BASIC では，数値同士については異なった型のデータ間で演算をすることが可能(自動的に型変換される)なので，明示的に型変換をしなければならないケースは比較的少なくなっています．ただし，数値と str 型との相互間においては明らかに取り扱いが異なるので，このための変換関数が用意されています(表2.2〜表2.4)．表中で，数値同士の変換関数のない部分は，単に相互間で代入(**左辺＝右辺**の形式)するだけですむので，あえて関数を設定していないのです．

一口に「変換する」といっても，そこには「意図」が含まれるので，同じ“5”でも，これを数値の5とみなすことによって int 型に変換するには **atoi** 関数を使います．もし，文字“5”のコードとして int 型に変換する場合は，**asc** 関数を利用します．asc 関数はあくまで文字コードという前提なので，変換前の文字数が複数のときは最初の1文字だけが対象となります．

数値とみなして float 型に変換する関数には，**val**，**atof** がありますが，両者とも考え方は同じです．

asc の逆関数は **chr$** で，この関数は文字コードを文字列(1字)に変換します．**itoa**，**str$** は，数値を10進表示文字列に変換するもので，2進，8進，16進表示としたいときは，それぞれ **bin$**，**oct$**，**hex$** を使います(表2.3)．

float 型から str 型に変換するときは，表2.4の **ecvt**，**fcvt**，**gcvt** のように戻り値以外にパラメータを経由して，関連情報も受け取るタイプがあります．

●表2．2　データ型変換関数一覧表

変換前のデータ型	変換後のデータ型			
	int	char	float	str
int	＼			ITOA その他＊1
char		＼		CHR $
float			＼	str $ その他＊2
str	ATOI ASC		{ ATOF VAL	＼

＊1は表2.3参照
＊2は表2.4参照

●表2．3　int 型から変換する拡張関数

変換前のデータ型	変換後の文字列形式		
	2進文字列	8進文字列	16進文字列
int	BIN$	OCT$	HEX$

●表２．４　float 型から変換する拡張関数

変換前のデータ型	戻り値は数値のみ		戻り値は実数または指数形式
	小数点以下桁数指定	文字列の桁数指定	
float	ECVT	FCVT	GCVT
第1パラメータ float	float 型変数		
第2パラメータ int	文字列の小数点以下の桁数	文字列の桁数	結果の桁数
第3パラメータ（結果） int	小数点の位置		
第4パラメータ（結果） int	符号 0＝正，1＝負		

第3章

画面表示

3-1 テキスト画面の設定命令

❖関連コマンド　**width, console**

テキスト画面は，X-BASIC の起動直後，64字×32行表示モードになっています．また，最下行の PF キー内容表示部分を除いて，全画面がスクロール画面として定義されています．

もし Human68K のコマンド入力時のように，字体を細目にしたいときは，

width | 64
96 |

●図 3．1　WIDTH によるモードの選択

(a)　96字モード

(0，0)

X 方向

Y 方向

32行

(95，31)

96字

(b)　64字モード

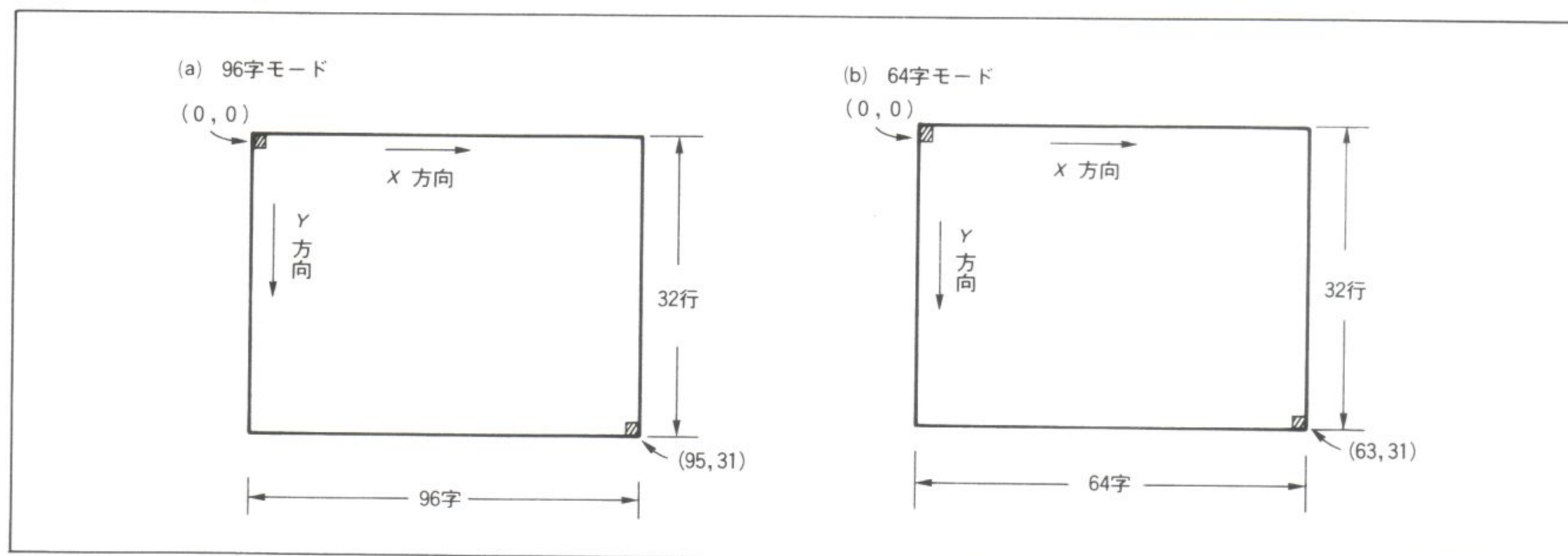
(0，0)

X 方向

Y 方向

32行

(63，31)

64字

画面表示のハードウェアについては，第Ｉ部で述べたとおりですが，本章では実際にX-BASICの命令によって表示，描画などを行なう際の命令について説明します．

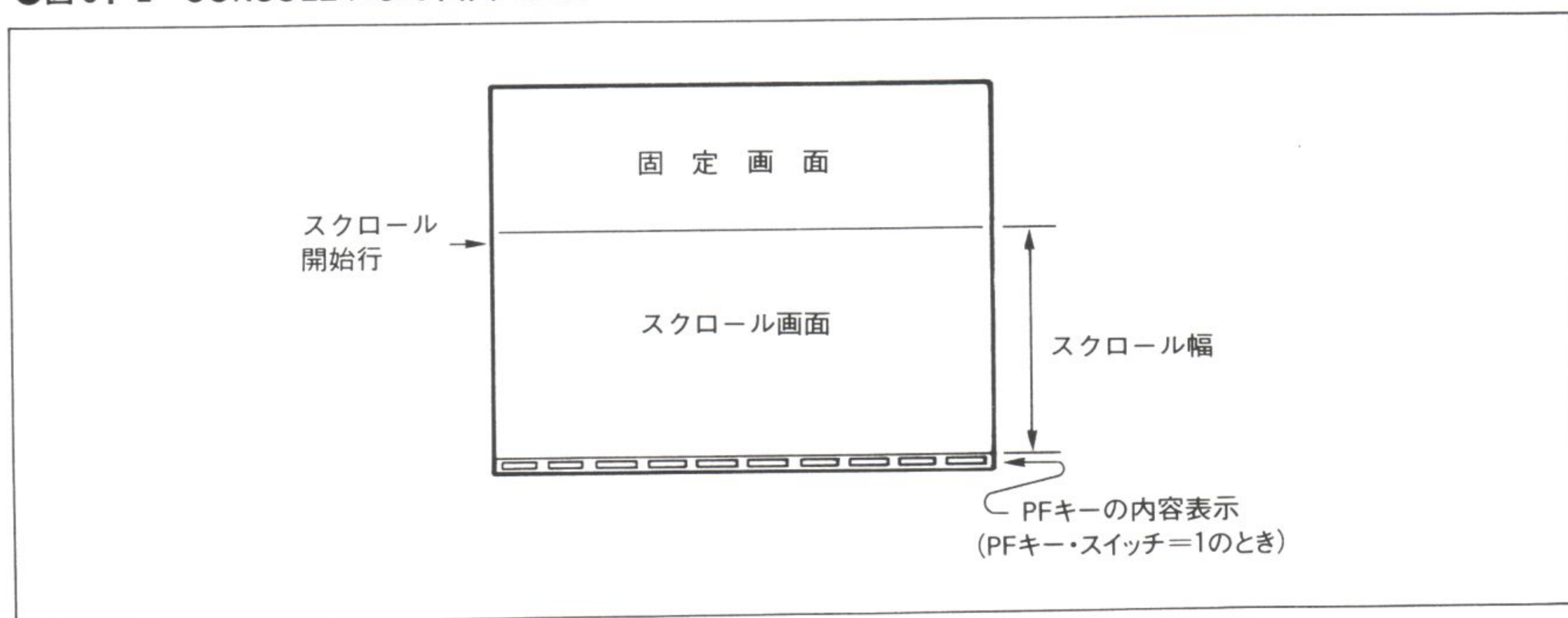
画面は，テキスト画面，グラフィック画面，スプライトを同時に表示できます．それに伴ない関連する命令もたくさん用意されていますが，ここでは画面の種類や用途に合わせて分類し，「交通整理」をしています．

コマンドで96字の側を選択すればよく，元に戻したければ64字を指定することでX方向の文字数を決めることができます(図3.1)．

ユーザ定義の入力画面がスクロールされては困る(言い換えると画面から以前の表示内容が消えていく)ときは，固定表示とスクロール表示の領域を区別する必要があります．このようなときは，

console 〈スクロール開始行〉，〈スクロール幅〉， $\left|\begin{matrix}0\\1\end{matrix}\right|$

●図３．２ CONSOLEによる画面の区別

固定画面
スクロール開始行
スクロール画面
スクロール幅
PFキーの内容表示
(PFキー・スイッチ＝1のとき)

コマンドで，スクロール範囲を限定します(図3.2)．このコマンドの最後のパラメータは，**PF キー**内容を表示するかどうかの**スイッチ**で，１ならば表示され，０ならば消されます．

つまり，スクロール画面であっても，実際，スクロールされるのは，今回表示する内容が最下行に表示され，さらに新しい表示内容が発生したときだけです．したがって，常に最下行を避けて表示するようにすれば，スクロールは行なわれません．ただし，以前の内容に重ね書きした場合，以前の内容は消されてしまいます．このような前提のもとで，スクロール画面内でも固定表示は可能です．しかし，ある部分は固定で，他の部分はスクロールしたいという場合は，console コマンドによる明示的な定義が必要です．

3-2 カーソル制御命令と関連システム変数

❖関連コマンド　　locate, csrlin, pos

カーソル位置は，print 命令などによって，スクリーンに文字が表示されるときに，自動的に決められます．このため，普通はカーソル位置をプログラムによって設定する必要性は少ないのですが，特別に設定した入力画面などでは，任意の位置にカーソルを点滅させたいことがあります．

このようなときは，

locate　<X 座標>, <Y 座標> 〔, ｜1/0｜ 〕

命令を使います．ここで，X 座標は横方向，Y 座標は縦方向の位置を表わします．続く省略可能なパラメータは，**カーソル・スイッチ**で，１ならば点滅，０ならば消します．省略すると現状を維持します．

カーソル位置を設定して，print 命令または input 命令で表示文字列を処理すると，その文字数だけカーソルは後方にズレます．改行コードがあったり，その行で表示しきれないときは，改行動作も伴ないます．その結果の位置に，次のカーソルが移動することを計算の上で，画面をデザインすることになるのです．

カーソル位置を決める上で，現在位置の座標を知りたいことがあります．その場合はシステム変数の

pos ………… X 座標
csrlin …… Y 座標

を利用して，目的の値を参照できます．これらの変数は参照のみで，代入によっては変更できません．

3-3 テキスト画面の消去，着色命令

❖関連コマンド　　cls, color, color[　]

BASIC プログラムで最も上の行から表示を開始したい場合や，ユーザ定義画面によって入力する場合など，それまで表示されていた全内容をクリアする必要が生じます．

cls

は，テキスト画面を消去する命令で，実行後画面が無表示状態になります．またスクロール画面についてはは最も上の行の先頭位置にカーソルを移します．

テキスト画面に表示する文字などに色を着けたいときは，

color　<色の具合>

命令を実行したあとに，print 命令で出力すると，以後変更されるまで指定された色具合に従って表示され

●表3．1　色具合を表わすコード

色＼状況	通常	強調	反転	反転強調
黒	0	4	8	12
シアン	1	5	9	13
黄	2	6	10	14
白	3	7	11	15

ます．色具合のコードは表3.1のとおりです．

また，パレット機能を使って，色の対応づけを変更したいときは，

color　[〈p0〉，〈p1〉，〈p2〉，〈p3〉]　（“[”，“]”はこのとおりに書く）

命令を使います．各パラメータは，パレットに対応するカラー・コードを指定するもので，p0は黒，p1はシアン，p2は黄，p3は白の各色のものを書きます．

3-4 グラフィック画面の設定命令

❖関連コマンド

screen, wipe, window, apage
home, vpage, contrast

グラフィック画面のモード設定は，

screen　〈表示画面サイズ〉，〈仮想画面・色モード〉，〈解像度〉，〈表示 ON/OFF〉

により行ないます．各パラメータの設定値は表3.2に従って与えます．

仮想画面のうち領域を指定するときは，

window (〈x1〉，〈y1〉，〈x2〉，〈y2〉)

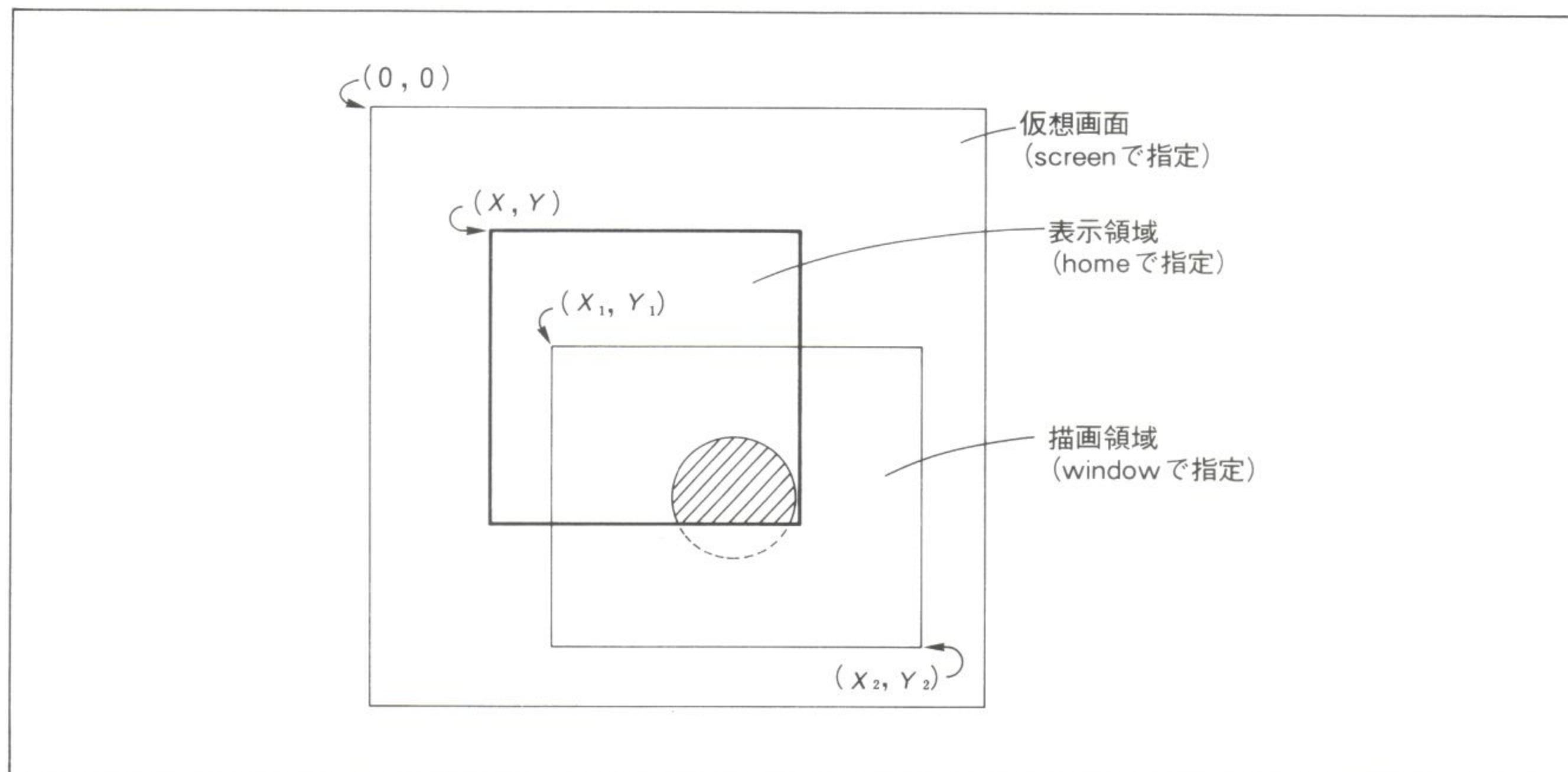
で対角線座標を与えます．表示範囲はこれとは直接関係せず，

home (〈ページ番号〉，〈x〉，〈y〉)

●表3．2　screen 命令のパラメータ

パラメータ	設定値	意味
表示画面サイズ	0	256×256
	1	512×512
	2	768×512
仮想画面，色モード	0	1,024×1,024，16色（ページ0）
	1	512×512，16色（ページ0～3）
	2	512×512，256色（ページ0，1）
	3	512×512，65,536色（ページ0）
解像度 (High/Low Res.)	0	Low（標準解像度）
	1	High（高解像度）
表示 ON/OFF	0	表示 OFF（グラフィック関数使用不可） ・グラフィック RAM をグラフィック以外の目的で利用 なお，いったん“1”に設定するとすべてのデータは消去される
	1	表示 ON（グラフィック関数使用可） ・グラフィック RAM 初期化

●図3．3　グラフィック画面設定命令の関連（太線枠内が実際に見える範囲）

(0，0)
仮想画面
(screenで指定)
(X，Y)
表示領域
(homeで指定)
(X_1, Y_1)
描画領域
(windowで指定)
(X_2, Y_2)

で指定されたポイントを左上とした表示画面サイズの範囲が対象となります(図3.3)．

screen を実行すると，window は，(0 , 0)を (x_1, y_1) として，表示範囲サイズに合った(x, y の対応ドット数からそれぞれ1を引いた値) x_2, y_2 の値に設定されます．また，home は(0 , 0)になります．ですから，これらの命令を単独で使うのは，screen による設定がすんだあとでなければなりません．

描画するページの指定は，

apage (<ページ番号>)

によって行ない，

vpage (<ページ ON/OFF コード>)

●表3．3　vpage 命令で指定するページ ON／OFF コード

i	グラフィックページ3	グラフィックページ2	グラフィックページ1	グラフィックページ0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

1……表示ON
0……表示OFF

スクリーン・モードによるiのとりうる値

1,024×1,024（16色モード）……0，1
512×512（16色モード）…………0から15
512×512（256色モード）………0から3
512×512（65,536色モード）……0，1

で表示するページの組み合わせを決めます．このときのコードは表3.3のようになります．

グラフィック画面でcls(クリア・スクリーン)に相当するのは，

```
wipe ( )
```

です．この命令は，現在描画対象になっている(apageで有効にされた)ページの内容をクリアするのに使われます．後のカッコを忘れやすいので，使うときには注意した方がよいでしょう．

なお，home以下のページ番号は，screenで指定した仮想画面モードで許されている範囲でなければならないことはいうまでもありません．

描画中の状況をスクリーンに表示したくない，つまり「舞台裏」を隠したいときは，別のページを表示させておく方法か，同じページでも表示範囲と描画範囲が重ならないようにする方法があります．

3-5 図形などを描く命令

❖関連コマンド　pset, line, box, circle, symbol

ここでは，描画関係の命令を取り上げます(図3.4)．

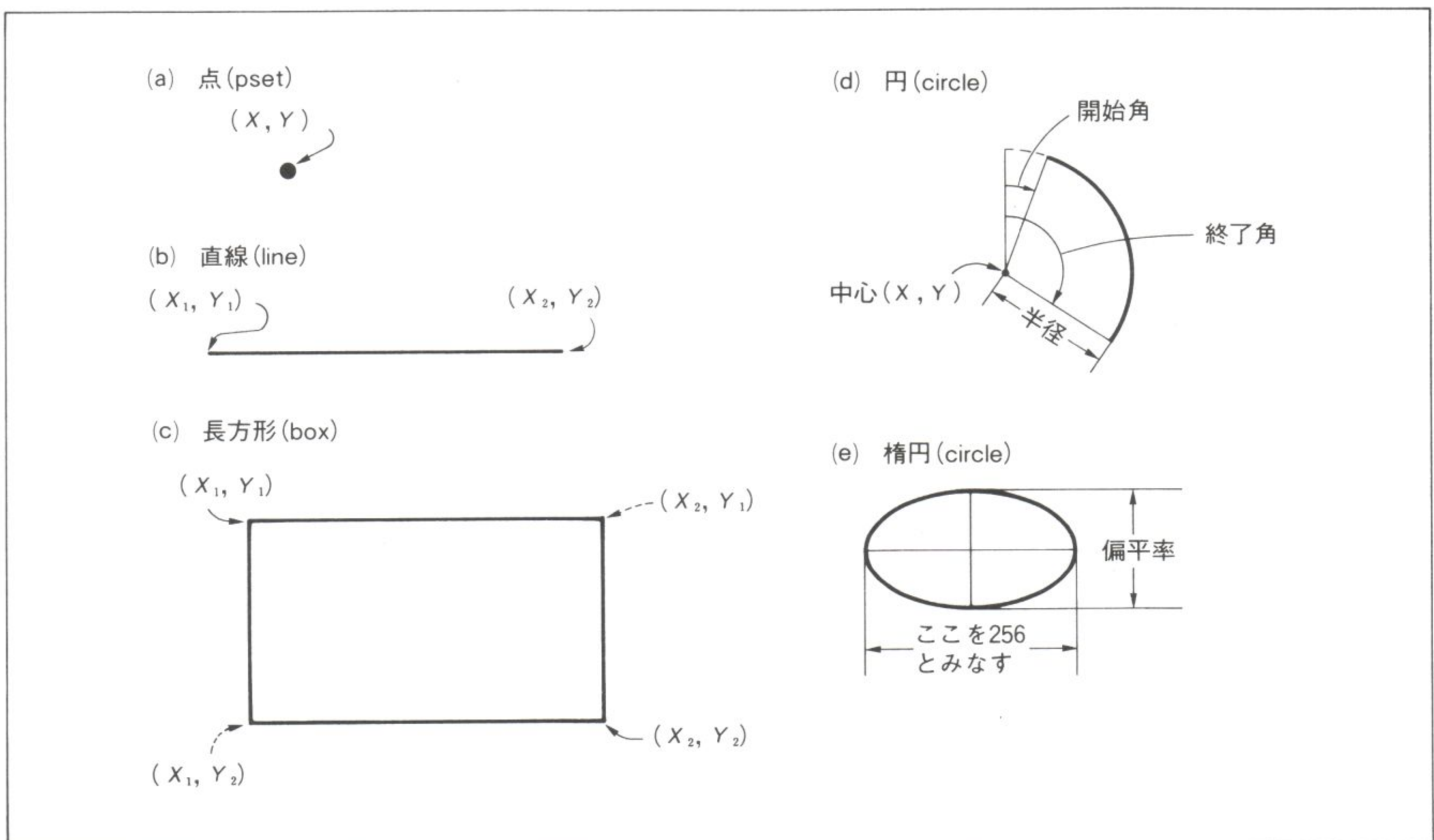
最も基本的なものは，画面上に点を描く，

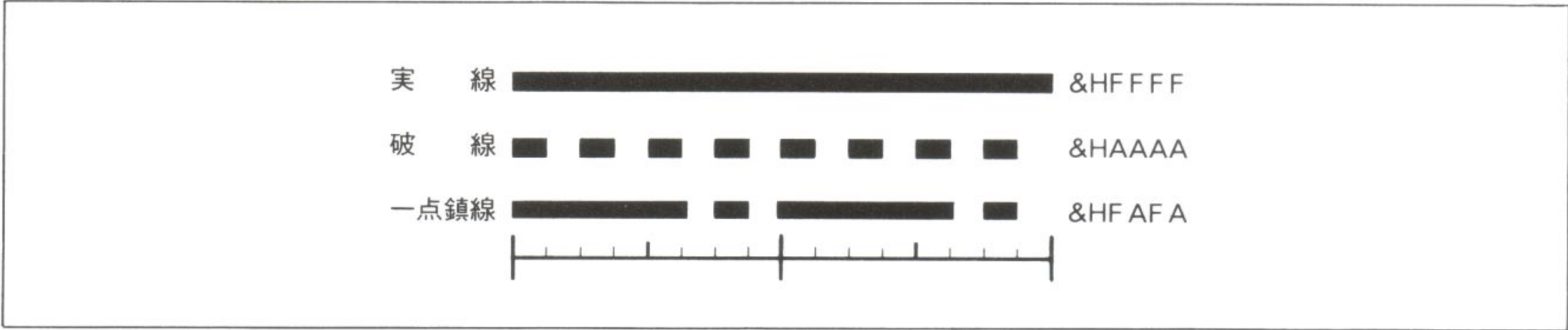
```
pset (<x>, <y>, <初期色*>)
```

です．この命令は，$(x,\ y)$で示される座標に，初期色で与えられる色の点を表示します(図3.4(a))．初期色は色モードの範囲内で指定され，パレットにより最終的な色が決定されます．

直線はpsetの連続でも引けますが，

```
line (<x1>, <y1>, <x2>, <y2>, <初期色> [, <ドット・パターン>])
```

●図3．4　描画命令とパラメータの関係

＊初期色をX-BASICでは「パレット・コード」と呼んでいる．

●図3．5　ドット・パターンと線の形状

を使うと一度に描けます(図3.4(b))．線は(x_1，y_1)から(x_2，y_2)の間に引かれますが，16ビットのドット・パターンを指定することによって，実線(&HFFFF)，破線(&HAAAA)，一点破線(&HFAFA)など任意の形状が選べます(図3.5)．

直線を利用すれば，三角形や四辺形が描けます．その中で，長方形と正方形は，

box (<x1>, <y1>, <x2>, <y2>, <初期色> [, <ドット・パターン>])

命令を使うと一度で描画が完了します(図3.4(c))．

また円はわざわざ座標を計算して pset するまでもなく，

circle (<x>, <y>, <半径>, <初期色> [, <開始角度>, <終了角度> [, <偏平率>]])

命令を使うことにより，楕円も含めて描けます．開始と，終了角度は，度数法(０～360)で与え，切り取られた孤の部分を作るのに指定します(図3.4(d))．省略時は輪の状態につながった形となります．偏平率は256のとき真円となり，これより小さい値は横長，大きい値は縦長の楕円を形成します．この値を256で割った値が縦横比です(図3.4(e))．

3-6 着色関係の命令

❖関連コマンド　palet, paint, fill, point, hsv, rgb

ここでは，描いた図形に対する着色関係の命令について述べます．

初期色などのカラー･コードを決める際には，直接数値で与えることもできますが，間接的な方法として次のようにもできます．１つは，

rgb (<赤の強さ>, <緑の強さ>, <青の強さ>)

で合成カラーコードを得るもので，各色の強さは０～31(５ビット)の範囲で与えます．結果は

５ビット×３色＝15ビット

から，残りの１ビットを０にして最下位に置くので，必ず偶数値となります．

もう１つは，合成カラーコードを得るのに，

hsv (<色相>, <彩度>, <明度>)

を使う方法です．色相は０～191(８ビット)，彩度と明度は０～31(各５ビット)ですが，結果はrgbと統一して，最下位に０を置いた16ビット値になります．これは，第１部で述べたハードウェア構成に合わせた形です．

rgb，hsvのいずれも関数で，結果は戻り値として得られます．

図形に着色するには，

paint (〈x〉, 〈y〉, 〈初期色〉)

で座標，色の指定を行なうことにより，着色が開始されます．注意しなければならないのは，座標は図形内のどの位置でもかまいませんが，図形は完全に線で囲まれている必要があります．1ヵ所でも破れがあると，そこから色が漏れ出し，関係のないところまで塗ってしまうことになります．

むしろ四角形の場合には，枠が作られていなくても四角形に塗ることのできる，

fill (〈x1〉, 〈y1〉, 〈x2〉, 〈y2〉, 〈初期色〉)

を使うのが簡単です．この命令は，box と paint を1つにしたようなものと考えることができます．fill で作った四角形は，paint で塗り直しすることも可能です．

スクリーンで見える最終的な色は，パレットによる変換結果で決まります．パレットの変換テーブルの要素は，

palet (〈初期色〉, 〈変換色〉)

で与えられ，初期色とこれに対応する変換色が1対1で指定されるようになっています．色変換は，16色などの少ない色をカムフラージュしたり，同じ色について画面全体にわたり瞬時に色替えするなどの効果を出したいときに利用されます．

着色に際しては，スクリーンの特定部分の色をコピーしたいことがあり，このようなときはその部分を参照するのに，

point (〈x〉, 〈y〉)

関数を使います．この関数は，戻り値として指定座標の初期色のコード値を与えるようになっています．

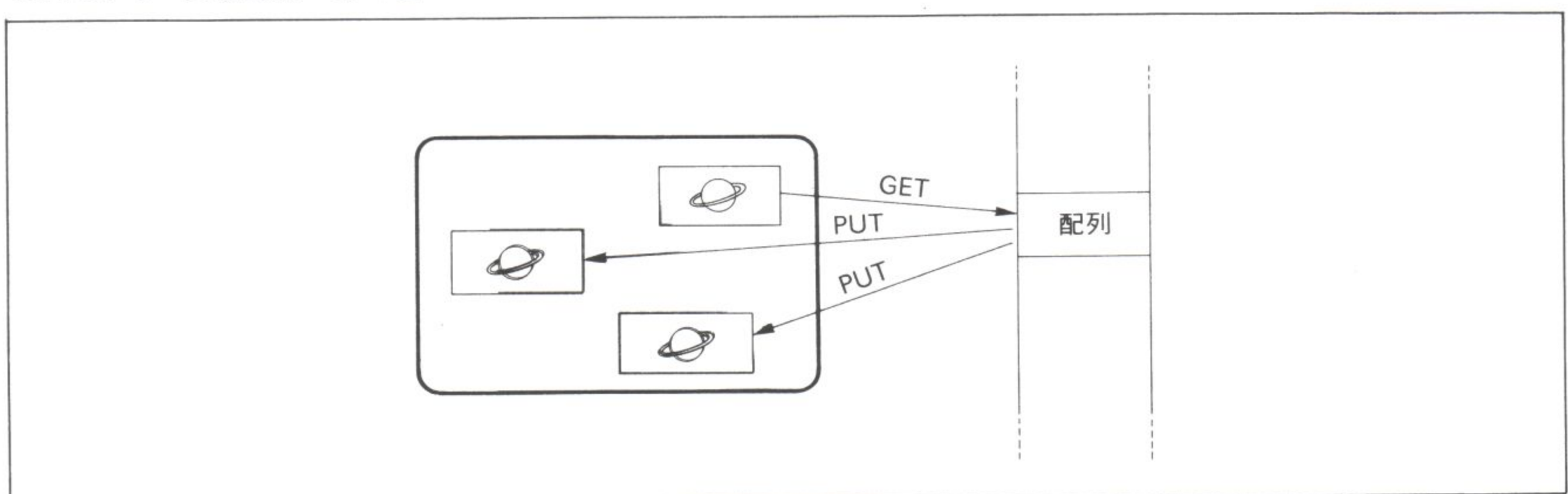
3-7 図形のコピー

❖関連コマンド　　　get, put

少し複雑な画面の描画においても，同じような図形が繰り返し出てくることは意外にたくさんあります．このような場合毎回演算を行なって描画するよりも，その図形のコピーを行なうほうが簡単で，かつ高速に処理できます．

コピーの際は，最初に

get (〈x1〉, 〈y1〉, 〈x2〉, 〈y2〉, 〈一次元数値配列名〉)

●図3.6　複数箇所へのコピー

で配列にひとまずコピーし，そこから目的の画面領域にコピーします．この場合，配列から画面領域への転送は，

put (<x1>, <y1>, <x2>, <y2>, <一次元数値配列名>)

命令を使います．したがって複数ヵ所へのコピーは，get 1回に対し put 数回というパターンになります(図3.6)．

このときの問題は，配列の大きさをどれだけ確保すればよいかという点ですが，16色モードでは4ビット，256色では8ビット，65,536色では16ビットが1ドットに対応することから，これらの値にドット数を掛けただけのトータル・ビット数が必要になります．そして，収容する配列のデータ型は，char 型では8ビット，int 型では32ビット，float 型では64ビット単位に個々の要素が配置されるので，これらの値で割って，小数以下の端数を切り上げた値が必要な要素数になります．

3-8 スプライト制御関係の命令

❖関連コマンド　**sp-clr, sp-color, sp-def, sp-disp sp-move, sp-off, sp-on, sp-pat**

グラフィック画面のうちでも，とくにスプライトはX68000のハイライトともいうべき装備です．ここでは，そのスプライトを，X-BASIC 上で扱う各命令の機能とそれぞれの関係というアプローチから解説します．

スプライトに対する描画関係では，パターンをクリアして透明にするところから始めるのが普通です．このための命令は，

sp_clr ([<開始スプライト番号> [, <終了スプライト番号>]

で，クリアするスプライトの範囲を指定します．終了スプライト番号省略時は，開始スプライト番号で指定されたものだけが対象となり，開始と終了スプライト番号の両方が省略されたときは0～255(全スプライト)がクリアされます．

パレットの設定は，

sp_color (<初期色>, <変換色>)

で各色の値を転送し，実際の描画は配列からドット・データをスプライトに転送する方法で行ないます．すなわち，

sp_def (<スプライト番号>, <配列名>)

命令で，前もって準備された配列と目的の番号のスプライトとを対応づけます．

反対に，スプライトに描かれている内容を配列にセーブしたいときは，

sp_pat (<スプライト番号>, <配列名>)

命令を使います．

スプライトに設定されたデータを表示するためには，128個ある**プレーン**(0～127)に張り付けなければなりません．ここでいうプレーンとはいわば優先度をもった仮想画面で，番号が小さいほど高優先(同じ座標位置にある場合，前面にあるように表示される)となります．

一方，“スプライト面”はすべてのプレーンをまとめた画面のことで，スプライト面を表示するかどうかのコントロールは，

sp_disp (<ON/OFF スイッチ>)

命令で行ないます．ON/OFF スイッチは表示 ON のとき１，OFF のとき０を指定します．

個別のプレーンは，

sp_off ([<開始プレーン> [, <終了プレーン>]])

で指定された範囲のものを表示からはずせます．終了プレーンを省略すると，開始プレーンのみが対象となり，どちらも省略すると全プレーンが対象になります．

反対に ON にするときは，１つには

sp_on ([<開始プレーン> [, <終了プレーン>]])

を使う方法があり，この場合範囲指定の意味は sp_off と同じです．

よく使われるのは，スプライトをプレーンに張り付けるか，または移動する，

sp_move (<プレーン番号>, [<x>], [<y>] [, <スプライト番号>])

命令です．この命令には，スプライト番号を指定すると自動的に該当プレーンを ON にする働きがあります．表示位置(x, y)は，省略すると以前の値と同じ座標値が採用され，移動しません．

第4章

一般の入出力命令

4-1　キーボードから入力する命令およびシステム変数

❖関連コマンド　　**input, linput, inkey$**

キーボードからのデータ入力に際しては，その形態によって異なった命令または，システム変数を使います．

最も一般的なものは，項目対応の入力です．この目的には，

```
input  ["〈表示文字列〉" | ; | ] 〈変数名〉 {, 〈変数名〉}
                         | , |
```

命令が適当です．この命令を実行すると，スクリーンには(指定があれば)表示文字列が出力され，その次の位置でカーソルが点滅して入力待ちになります．入力データは，“,”で区切って対応する順序に入力し，最後に[↵]で終了させます．なお表示文字列の次に“；”を指定したとき，または表示文字列全体を省略したときは，スクリーンの対応位置に“？”が表示されます．

この場合，文字列に“,”などの文字を含むとそこで切られ，あとは別な項目とみなされます．このため，そうなっては困るときは，引用符で囲むという方法があります．しかし，行対応でテキスト文などの入力に使う．

```
linput  ["〈表示文字列〉"；] 〈文字変数〉
```

命令によれば，“,”や引用符も含めた入力が可能になります．ただし，この命令の場合，文字変数は１命令について１つしか入力できません．

以上いずれの命令も，[↵]で終了するため，“[↵]”そのものは入力できません．これは，文字列を入力する場合の宿命のようなものです．

本章では，スクリーン以外の入出力命令について説明します．

対象となるのは，キーボード，プリンタ，ブザー，ディスク，FM 音源，PCM 音源，マウス，ジョイスティックです．各デバイスの詳細については，第 I 部の説明も併せて読んでください．

FM 音源のような命令の関係が複雑なデバイスについては，命令の関連図を掲載するなど，工夫してあります．

この点で，どんな文字にも対応できるようにするには，システム変数 inkey$ を使います．この変数は，キーボードから 1 文字入力されるまで待ち，その入力された文字を str 形式で代入します．マイクロソフト系 BASIC では，入力待ちがないので，キーが押されていないときは空文字が代入されるという機能があります．しかし，X-BASIC では必ず 1 字が代入される点に注意しなければなりません．しかし，逆に考えるとカーソル制御をしなくてもよい点は，X-BASIC の使いやすいところといえます．

4-2 PF キー関係の命令およびコマンド

❖関連コマンド　key, keylist

PF キーは，特定の文字の並びを登録しておき，キーを押したときにそれを一度に取り出したいときに用いられます．

PF キーの数は，[F 1]～[F 10]の10個ですが，F-BASIC では[SHFT]+[Fn]の方法で PF11～20にも対応できるようになっています．登録は 1 つの PF に対し，

key 〈番号〉,〈登録文字列〉

のように行ないます．いちいち手動で入力するのも大変なので，一般には複数キー分をまとめてプログラ

ムにし，自動的に設定する方法がとられます．

文字列の中には ↵ などのコントロール・コードを含みたい場合があるので，このようなときは，対応する CTRL ＋アルファベットの関係を調べ，たとえば， ↵ (CTRL ＋M)の場合@Mのように書きます．また，

```
load "prog"
```

のように引用符を使いたい場合もあり，このようなときは引用符は@を２つ続けることで代替できます．この例では，登録文字列を，

```
load @@prog@@
```

と定義します．

今現在の PF キーの登録内容を参照するには，

```
key list
```

コマンドを使います．

4-3 プリンタへの出力およびブザーを鳴らす命令

❖関連コマンド　　**lplint, llist, lfiles, beep**

本来 print 命令は，端末装置のプリンタで出力する命令だったのですが，パソコンではプリンタよりディスプレイ装置の普及が先に進んだため，スクリーンに出力する命令に変わってしまいました．このため，本来の print に相当する命令として lprint ができ，これをプリンタに出力する命令として使うようになりました．

同様に，スクリーンに表示する list，files 命令についても，プリンタ用に llist，lfiles が作られました．

以上の各命令は，出力先がプリンタに変わるだけで，文法はまったく同じです．これらの文法については，各命令("l" なし)の説明を参照してください．

X-BASIC で筆者が残念に思うのは，print などの命令がデバイスを特定していることです．もし OS-9 の BASIC のように標準出力になっていれば，コンパイル後スクリーンでもプリンタでも，さらにはディスク・ファイルでも自由に接続できるので，lprint のような命令はいらなくなるのです．

ところで，input 命令などでの入力ミスに対しては，エラー・メッセージを出すと同時に警報音を出すことがよく行なわれます．また，パソコンに何かの仕事をやらせっ放しにする場合，終了したとき「ピッ，ピッ，ピッ」というような合図を出すようにプログラムを組んでおくと，いちいち画面を見に行く手数が省けます．

このようなとき，beep 命令を使うか，文字コード＆H07(Bell)を表示(print)させることによって，ブザー音を発生させることができます．

単発で鳴らすときは簡単ですが，複数回連続して鳴らすときは，時間調整が必要です．このために内容のない for～next ループを使って時間を消費したり，time＄の値を監視して変化したときだけ beep させる(１秒ごとに鳴る)などの方法がとられます．

4-4 ディスク・ファイルの入出力関数

❖関連コマンド　**fopen, fread, freads, fgets, fseek, fwrite fwrites, fputc, feof, fclose, fcloseall**

ディスク・ファイルに入出力をするときは，最初の手続きとして，

fopen (〈ファイル名〉，〈モード〉)

を実行しなければなりません．このときのモードは，表4.1のように指定します．この関数の戻り値は，ファイル番号として以降の処理でも使用するので，ファイル処理終了まで保持しなければなりません．

ファイルを読むときは，その扱い方によって関数が分かれています．すなわちテキスト・ファイルのような文字列を書いてあるファイルの場合は，

freads (〈文字列名〉，〈ファイル番号〉)

を使います．この関数では，１行分のデータが読み込まれ，そのときの文字数が戻り値となります．

また，配列型データのときは，

fread (〈数値型１次元配列名〉，〈件数〉，〈ファイル番号〉)

により，ファイルから読み出すデータの件数を指定すると，相当分のデータが配列に埋められます．

ファイルから１字ずつデータを読み出したいときは，

fgetc (〈ファイル番号〉)

で，戻り値によって結果の値を受け取ります．

ファイルはデータが終了していると，読み込みを続行できなくなるので，

feof (〈ファイル番号〉)

関数の戻り値を常に注意していなければなりません．また，上記の読み込みのための関数では，ファイル終了が検出されたり，エラーがあると戻り値は－１になります．

なお，ファイルのデータをバイト単位で任意の位置から取り出したり，書き換えたりするときは，

fseek (〈ファイル番号〉，〈バイト相対位置〉，〈シーク・モード〉)

により位置づけします．この関数の戻り値には，処理後の新しいデータ・ポインタの値が入ります．このとき，もしエラーが発生すると－１となります．シーク・モードは基点となる場所を，０＝ファイル先頭，１＝現在位置，２＝ファイルの終わりのいずれかで指定します．

データを出力するときは，それがテキスト文のような文字列ならば，

●表4．1　fopenのモード

モード	意味
"r"	作成ずみのファイルの読み込み．ファイルがないときはエラーとなる．
"c"	新規，既存ファイルを問わず出力するのに適する．出力した内容は読むこともできる．
"w"	既存ファイルに出力する場合に指定する。ファイルがないときはエラーとなる．
"rw"	既存ファイルに入出力する場合に指定する．ファイルがないときはエラーとなる．

●図4．1　ソース・プログラム・ファイルの内容を66行ずつプリンタに出力するプログラム
（ファイル名はキーボードから入力する）

```
 10 str fn$[20]
 20 str ln$[80]
 30 str dummy$[1]
 40 int inf
 50 input "File Name=",fn$
 60 fopen(fn$,"r")
 70 while feof(inf)<>-1
 80   freads(ln$,inf)
 90   lprint ln$
100   cnt=cnt+1
110   if cnt=66 then input "OK? ",dummy$:cnt=0
120 endwhile
130 fclose(inf)
140 end
```

fwrites (**<文字列名>，<ファイル番号>**)

を使います．また，数値型配列ならば，

fwrite (**<配列名>，<書き込み件数>，<ファイル番号>**)

によります．１字ずつ出力する場合は，

fputc (**<char 型変数名>，<ファイル番号>**)

関数を利用します．それぞれの戻り値は，実際に書き込んだバイト数，書き込んだ件数，書き込んだ char 値が入り，エラーの場合は－１となります．

ファイルの処理がすんだら，

fclose (**<ファイル番号>**)

によって終了手続きを行ないます．プログラムの終わりなどで全部のファイルをまとめてクローズするには，

fcloseall ( )

を使えば一度ですみます．戻り値は，前者では０，後者の場合クローズしたファイル数が入り，エラーが発生したときは前者も含め－１となります．

以上の関数の応用例として，図4.1に BASIC ソース・プログラム・ファイルの内容を66行単位に分けて，プリンタで出力するプログラムを掲げます．このプログラムは，普通紙(A4)にプログラム・リストを出力するために作成したものです．

4-5 FM音源関係の命令

❖関連コマンド　**m_alloc, m_assign, m_cont, free, m_int, m_play m_stat, m_stop, m_tempo, m_trk, m_vget, m_vest**

FM音源で音楽を演奏するためには，たくさんの命令を使わなければなりませんが，マニュアルなどの個別の命令の説明を読みながら，全体の関連を理解しようとするのは骨が折れるものです．そこで，ここでは図4.2に示す関連図をもとに，個々の命令のつながりを解明したいと思います．

演奏しようとするとき，最初にしなければならないのは**トラック**の用意です．このためには，

m_alloc (**〈トラック番号〉，〈トラック・サイズ〉**)

命令でメモリを確保するのですが，この領域にはいわゆる「楽譜データ」が書き込まれます．トラックは，m_alloc と，FM音源を初期化する

m_init ( )

でクリアされます．

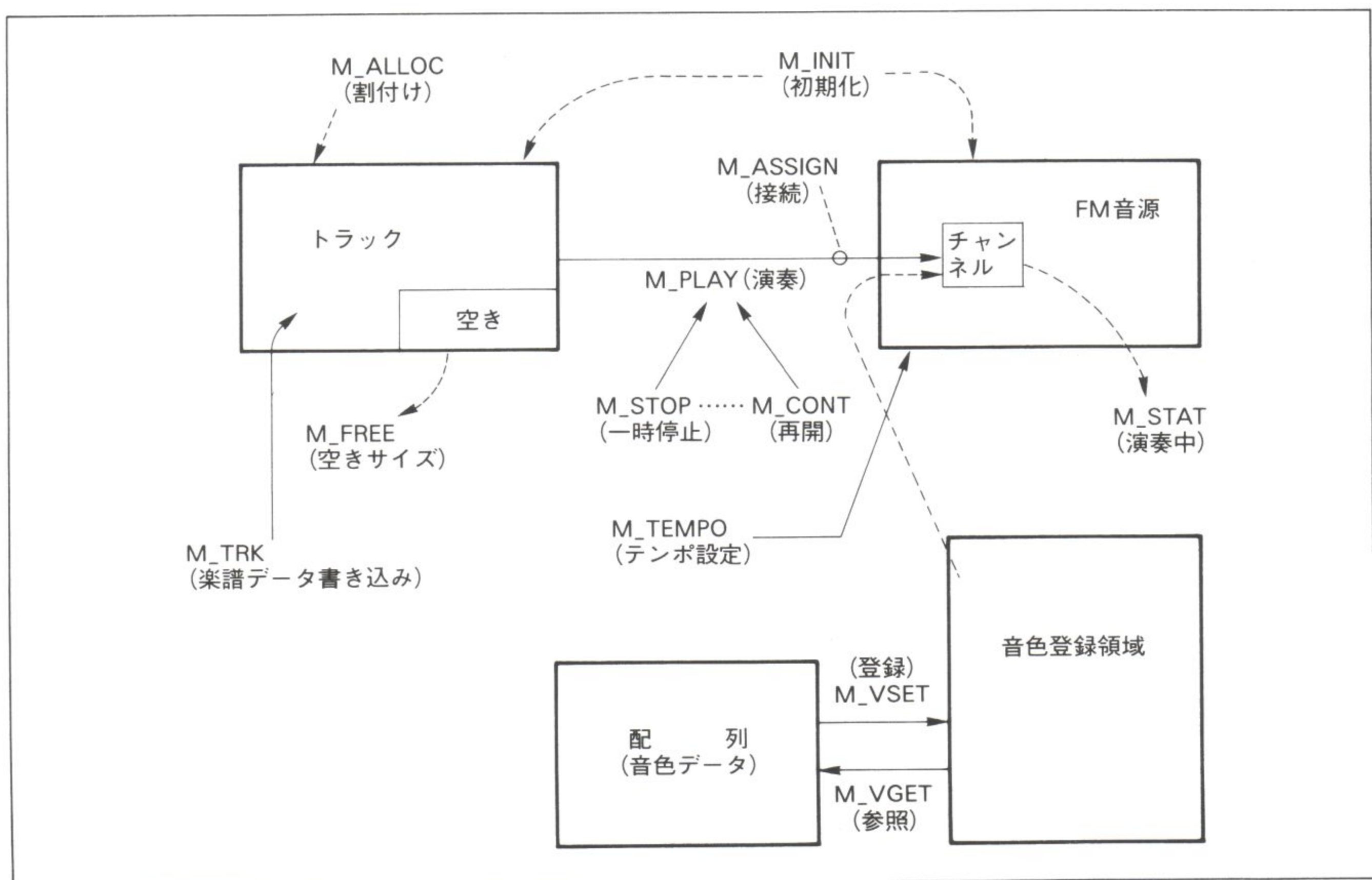
楽譜データは**MML**(ミュージック・マクロ言語)で書かれますが，この内容はプログラム中で文字列として定義されます．そして，トラックへの転送で演奏待機状態となります．このときの命令は，

m_trk (**〈トラック番号〉，〈MML文字列〉**)

を使います．

演奏を開始するためには，トラックとFM音源用のチャンネルとの関連づけが必要で，

m_assign (**〈チャンネル番号〉，〈トラック番号〉**)

●図4．2　FM音源操作命令の関連図

●表4．2　音色番号と楽器の種類

音色番号	種　類	音色番号	種　類	音色番号	種　類	音色番号	種　類
1.	A. PIANO	18.	ACCORDION	35.	HORN	52.	TRIANGLE
2.	H. PIANO	19.	VIOLIN	36.	TROMBONE	53.	COWBEL
3.	E. PIANO	20.	CELLO	37.	TUBA	54.	TUBLER BELL
4.	CLAVINET	21.	STRINGS1	38.	BRASS1	55.	STEEL DRUM
5.	CELESTA	22.	STRINGS2	39.	BRASS2	56.	GLOCKEN
6.	CEMBALO	23.	PIZZICATO	40.	HARMONICA	57.	VIBRAPHONE
7.	A. GUITAR	24.	VOICE	41.	OCARINA	58.	MARIMBA
8.	E. GUITAR	25.	CHORUS	42.	RECORDER	59.	H-H CLOSE
9.	W. BASS	26.	GRASS HARP	43.	SAMBA WHISTLE	60.	H-H OPEN
10.	E. BASS	27.	WHISTLE	44.	PANFLUTE	61.	CYMBAL
11.	BANJO	28.	PICCOLO	45.	SNARE DRUM	62.	SYN LEAD1
12.	SITAR	29.	FLUTE	46.	RIMSHOT	63.	SYN LEAD2
13.	HARP	30.	OBOE	47.	BASSDRUM	64.	AMBULANCE
14.	KOTO	31.	CLARINET	48.	TOM-TOM	65.	STORM
15.	P. ORGAN1	32.	BASSOON	49.	TIMPANI	66.	LASER GUN
16.	P. ORGAN2	33.	SAXPHONE	50.	BONGO	67.	GAME SE1
17.	E. ORGAN	34.	TRUMPET	51.	TIMBALES	68.	GAME SE2

命令により接続を行なわなければなりません．また，FM音源に対しては，必要があれば

　　m_tempo (〈テンポ〉)

で32～200のうちからテンポを設定します．この値は，全チャンネルに有効です．

準備が整い，演奏を開始するには，

　　m_play (〈チャンネル番号〉, {, 〈チャンネル番号〉})

でチャンネルを指定します．ここで( )内を省略すると全部のチャンネルが対象となります．

演奏は，トラックというバッファからチャンネルに対する「指令」を1つずつ取り出して，FM音源を制御することによって行なわれます．この中で音色の設定などは大事なパラメータで，MMLでは音色登録領域に展開されている音色データの番号(表4.2参照)を指定して，細かな音色情報をもらい受けてFM音源に設定します．

この音色登録領域の内容は，ユーザが配列中に設定した音源データ(形式は表4.3のとおり)を，

　　m_vset (〈音色番号〉, 〈配列名〉)

で別途追加登録することができます．反対に，任意の登録内容を参照したいときは，

　　m_vget (〈音色番号〉, 〈配列名〉)

で，配列に引き出すことができます．この命令の最終的な目的は，登録内容を一部アレンジして利用するところにあります．

さて，演奏中に何かの都合で一時停止したいときは，

　　m_stop (〈チャンネル番号〉, {〈チャンネル番号〉})

命令でチャンネルを指定します．( )内省略時はm_playのときと同じく，全チャンネルが対象となります．再開するときは，

　　m_cont (〈チャンネル番号〉 {, 〈チャンネル番号〉})

で再開チャンネルを指定すると，そのチャンネルの演奏が続行されます．( )内省略時は，全チャンネル再

●表4．3　音色データの配列形式

	0		1～4（オペレータ）	
0	フィード・バック/アルゴリズム	(0～63)	AR	(0～31)
1	スロット・マスク	(0～15)	D1R	(0～31)
2	ウエーブ・フォーム	(0～3)	D2R	(0～31)
3	シンクロ	(0,1)	RR	(0～15)
4	スピード	(0～255)	D1L	(0～15)
5	PMD	(0～127)	TL	(0～127)
6	AMD	(0～127)	KS	(0～3)
7	PMS	(0～7)	MUL	(0～15)
8	AMS	(0～3)	DT1	(0～7)
9	L,RPAN	(0～3)	DT2	(0～3)
10			AMS イネーブル	(0,1)

開となります．

トラックの内容は，演奏ずみのものが消されていくので，空きを利用して書き足すことによって長い曲の演奏が可能です．空き具合を見るには，

m_free (<トラック番号>)

の戻り値で残りバイト数を取得すればよく，次のMML文字列を収容できる分だけ空いた時点でm_trkで送り込むようにします．

FM音源が演奏状態にあるかどうかは，

m_stat (<チャンネル番号>)

で戻り値が1ならば演奏中，0ならば休止中と判断できます．全チャンネルの状態を一度に参照するときは( )内を省略すればよく，このとき戻り値はビット番号に1を加えた値がチャンネル番号に対応した情報になります．

4-6 MML（ミュージック・マクロ言語）

m_trk命令でトラックに書き込むMML文字列の個々の要素は，次のとおりです．

●テンポ(T*n*)

32から200の範囲でテンポを指定するもので，*n*は1分間の4分音符の回数を表わします．省略すると120が取られます．この値は，演奏するチャンネルのみに有効です．

●音量(V*n*)

チャンネルの音量を，0～15の範囲で設定するものです．省略値は*n*＝8となります．

●音程(A～G，＋，#，－)

Cをドとしたドレミの音階を表わします．音程文字の後に#または＋を付けると半音高く，－を付けると半音低くなります．

●休符(R)

休符は，音を出さず時間だけあける操作です．

●音長(n,.)

音程，休符の後に付けて，音長を表わします．*n*＝1が全音符で，音長は1／*n*に対応します．日本語でいう「*n*分音符」に相当します．省略すると，規定値(通常4)がとられます．最大値は64です．“.”は符点で，基本となる音長に5割加算されます．

●音長省略値(L*n*)

音長の規定値(通常4)を変更したいときに指定します．

●オクターブ（O*n*，＜，＞）

省略時のA（ラ）の音は440Hz（*n*＝4）で，これを基準として，変更したいときに0～8の範囲で指定します．単に1オクターブ上げるだけならば＞，1オクターブ下げるときは＜も使えます．

●音色（@*n*）

音色番号を，表4.2に従って指定するのに使います．省略値は*n*＝1です．

●タイ（&）

タイ記号に相当するもので，音が連続します．

●発音割り合い（Q*n*）

1音中で実際に音を出す割り合いを1～8で設定します．*n*＝1を1/8として，8/8まで指定することになります．小さいほど歯切れのよい音になり，大きいほどソフトな感じが得られます．省略値は8です．

●連符（{*m*} *n*）

*n*分音符を*m*等分した音長を得たいときに使います．割り算した結果が64分音符より短いと，エラーになります．また*m*の値が2の整数乗からはずれ，かつ大きいときは誤差が累積されて，他のチャンネルと同期がとれなくなることがあります．

●音量微調整（@V*n*）

音量を設定する際，細かく指定したいときに128段階（0～127）で記述できるものです．すでにV*n*で設定されていても，V*n*の値で置き換えられます．

●音長微調整（@L*n*）

音長規定値を，192段階（1～192）で指定するときに使います．L*n*で指定してあっても，後で指定したほうがその時点から有効になります．注意が必要なのは，192が全音符に対応し，*n*の値を192で割った値が音長になるという点です．したがってこの値はL*n*とは反対に，大きいほど長くなります．4分音符（L4）に対応する値は@L48です．

●繰り返し（|：*n*～：|）

同じ演奏内容の繰り返しには，|：と：|の間にその内容を記述します．*n*は繰り返し回数で，省略すると2がとられます．：|に到達して指定回数に満たないときは，|：に戻るのが繰り返しの仕組みです．

●繰り返し内容の例外（|*n*）

繰り返し演奏する中で，ある回だけ別な内容にしたいときは，|*n*に続いてその内容を指定します．その回で終了しないときは，：|で|：まで戻らせるようにしなければなりません．

●レジスタの直接設定（Y*n*，*d*）

システムのFM音源駆動機能に不満足なマニア向けの機能です．*n*でレジスタ番号を指定し，直接データ（*d*）を書き込みます．レジスタの内容その他は第1部を参照してください．

●W*n*

レジスタの直接設定により，.KEY ON／OFFした直後に有効な，音長に相当する時間を確保するコマンドです．*n*は音長で説明した内容と同じで，前もってL*n*によって設定されていなければ，省略値は4となります．

4-7 PCMデータの録音再生

❖関連コマンド　**a_play, a_rec**

PCM関係の機能は録音と再生だけで，しかもメモリのテーブルを利用して行なうため，プログラムは非常に簡単です．

録音の際は，

a_rec (<配列名>, <サンプリング周波数コード>)

命令で，一次元配列とサンプリング周波数(表4.4)を指定します．周波数は高いほどクリアな音になりますが，録音できる時間は短くなります．

再生時には，

a_play (<配列名>, <サンプリング周波数コード>, <出力モード>)

命令を使います．ここで，サンプリング周波数は録音時と同じ値を指定しないと，再生された音の周波数が変わってしまいます．出力モードについては，表4.5のとおりですが，ステレオの場合は左右同じ音が出るだけで，立体音にはなりません．

●表4．4　サンプリング周波数コード

コード	サンプリング周波数
0	3.9 KHz
1	5.2 KHz
2	7.8 KHz
3	10.4 KHz
4	15.6 KHz

●表4．5　出力モード

出力モード	接続
0	出力しない
1	左のみ
2	右のみ
3	ステレオ

4-8 マウス関係の命令

❖関連コマンド　**mouse, msarea, msbtn, mspos, msstat, setmspos**

マウスを使用するときは，スクリーン上のマウス・カーソルを利用するのが通例です．その場合，必要があればカーソルの移動範囲を，対角線座標により，

msarea (<x1>, <y1>, <x2>, <y2>)

命令で制限します．

マウスの初期化やカーソルの表示，消去は，

mouse (<動作コード>)

命令により，動作コードを0＝初期化，1＝カーソル表示，2＝カーソル消去，3＝カーソル表示の有無を戻り値(表示中なら－1，それ以外は0)で得るのいずれかで指定します．

カーソルが表示中ならば，マウスが移動しているときはカーソルも連動してスクリーン上を移動します．カーソルの動いた跡をそのまま描画するには，

mspos (<x>, <y>)

によってX，Yに相当する変数に座標を取り込み，psetなどで対応位置のグラフィック画面に表示すればよいのですが，その際には頻繁にmsposとpsetを繰り返すことになります．

描画するしないにかかわらず，操作者がプログラムの要求する特定座標などの情報を与えるには，ボタンを利用します．ボタンの状態を参照するには，

msbtn (**<ボタン待ちの状態>，<左右>，<待ち時間最大値>**)

関数で待ち合わせも兼ねるのが普通です．ここで，ボタン待ちの状態は１ならばボタンが押されるまで，０ならば離されるまでの指定となり，その状態になったときこの命令の実行が終了します．左右はどちらのボタンかを示すもので，０が左，１が右を表わします．待ち時間の最大値は，０のときは無限大となり，ボタン待ちの状態が満足されるまで待ち合わせが行なわれます．０以外のときは，大きな値を指定するほど長くなります．この関数では，戻り値として指定された状態になるまでの時間カウントが得られます．

一方，カーソル位置，ボタンは，

msstat (<Δx>, <Δy>, **<左ボタンの状態>，<右ボタンの状態>**

命令で参照することもできます．Δx, Δyは以前に参照したときからの相対移動量を表わす変数を指定し，－128～127の値を受け取ります．左右のボタンの状態を表わす変数には，ONならば－１，OFFならば0がセットされます．

最初のマウス・カーソルを任意の位置に設定するには，プログラムで対応せざるを得ないので，このためには，

setmspos(<x>, <y>)

命令を使います．

マウスを扱うプログラムでは，現時点の座標(x，y)を一元的に管理し，各命令実行時に矛盾が起きないようにしなければなりません．

4-9 ジョイスティック関係の命令

❖関連コマンド　　stick, strig

ジョイスティックは，２系統使えるようになっており，それぞれ１番，２番として区別しています．

ジョイスティックの特徴は，単純な方向のみを表わすデータを入力するところにあり，

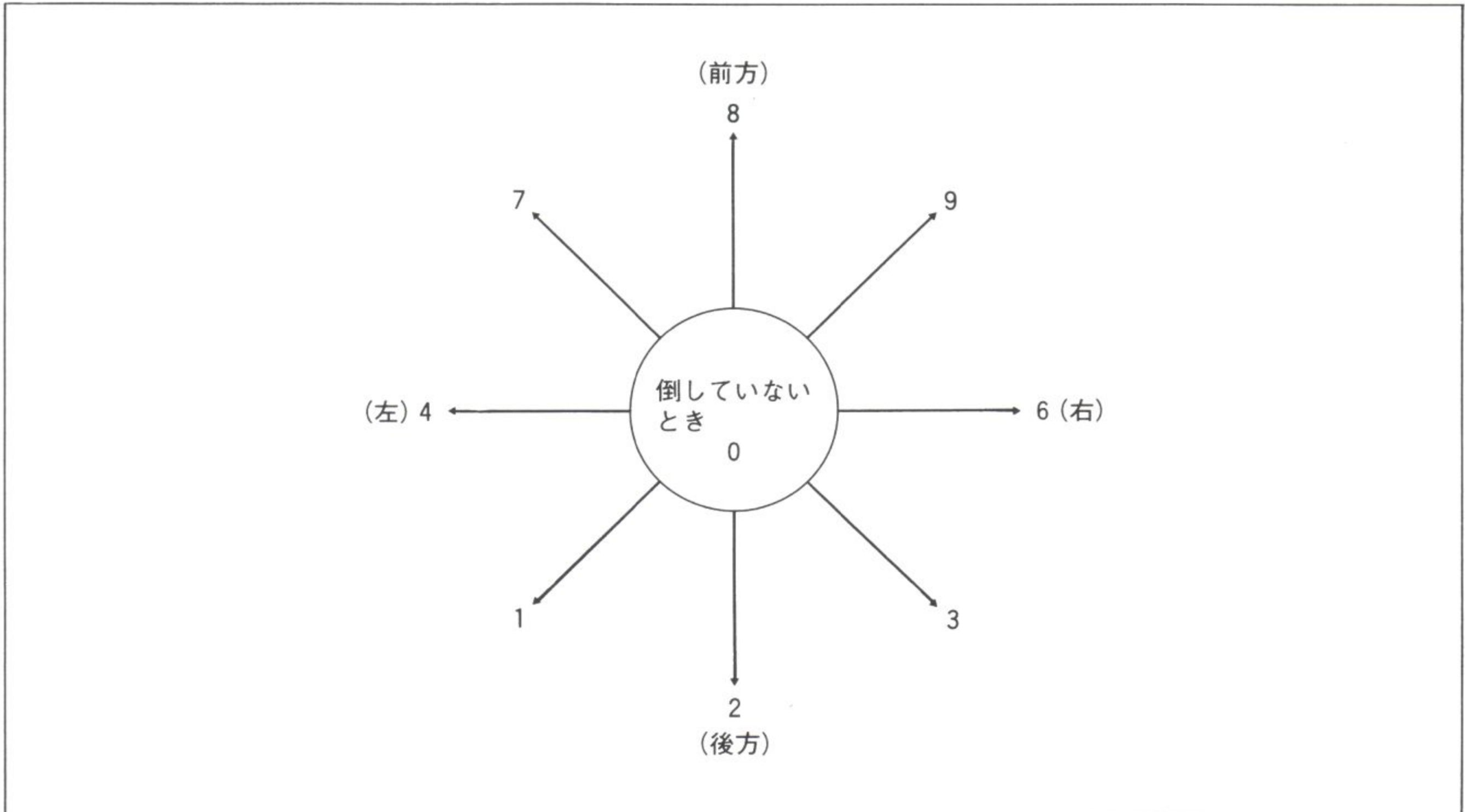
stick (**<番号>**)

関数で指定番号ジョイスティックの状態を参照する(戻り値を得る)ことができます．戻り値は図4.3のように，スティックが倒されていないとき０，それ以外は８方向に対応した値が得られます．

そのほかに，ジョイスティックには補助的な役割りをもったトリガ・ボタンが2個用意されており，この状態は，

strig (**<番号>**)

関数で参照できます．戻り値は，トリガ・ボタン１が1，トリガ・ボタン２が2として，押されているボタンの値を合成した値が得られます．たとえば，3ならば両方がONで，0ならばいずれもOFFということになります．

●図4．3　ジョイスティックの方向と戻り値

第5章

応用プログラムとC変換，外部関数の作成

5-1 タブ処理を行なうBASICプログラム

ここでは，極めて実用的なBASICプログラムのサンプルを紹介します．このようなプログラムは，アセンブラでプログラムを作成する際に必需品ともなるものです．

図5.1は，エディタで作成したソース・プログラムの原形です．このソースをX-68000のアセンブラで処理すると，図5.2のようなリストが出力されます．このままではラベル付きの命令とそうでないものとの命令記述が不揃いで，見苦しいことこの上もありません．タブを使って入力すればよいのですが，かと言ってすでにタブなしで入力したものをエディタで揃えるのも大変なので，プログラムを使って位置揃えを行なうことを考えます．

方法は，図5.3のように，スペースで区切られた各ブロックを該当する位置に配置するもので，例外的に

●図5．1 TABキーを使わずに入力したソース・プログラムの例

```
*
*     None Macro
*

 include ¥include¥doscall.mac

start:
 pea text_line *Set text address
 dc.w  _PRINT *DOS-call
 addq.l #4,sp *Release stack
 dc.w _EXIT *End of program
*
text_line dc.b 'Test ok',$0d,$0a,0
 end start
```

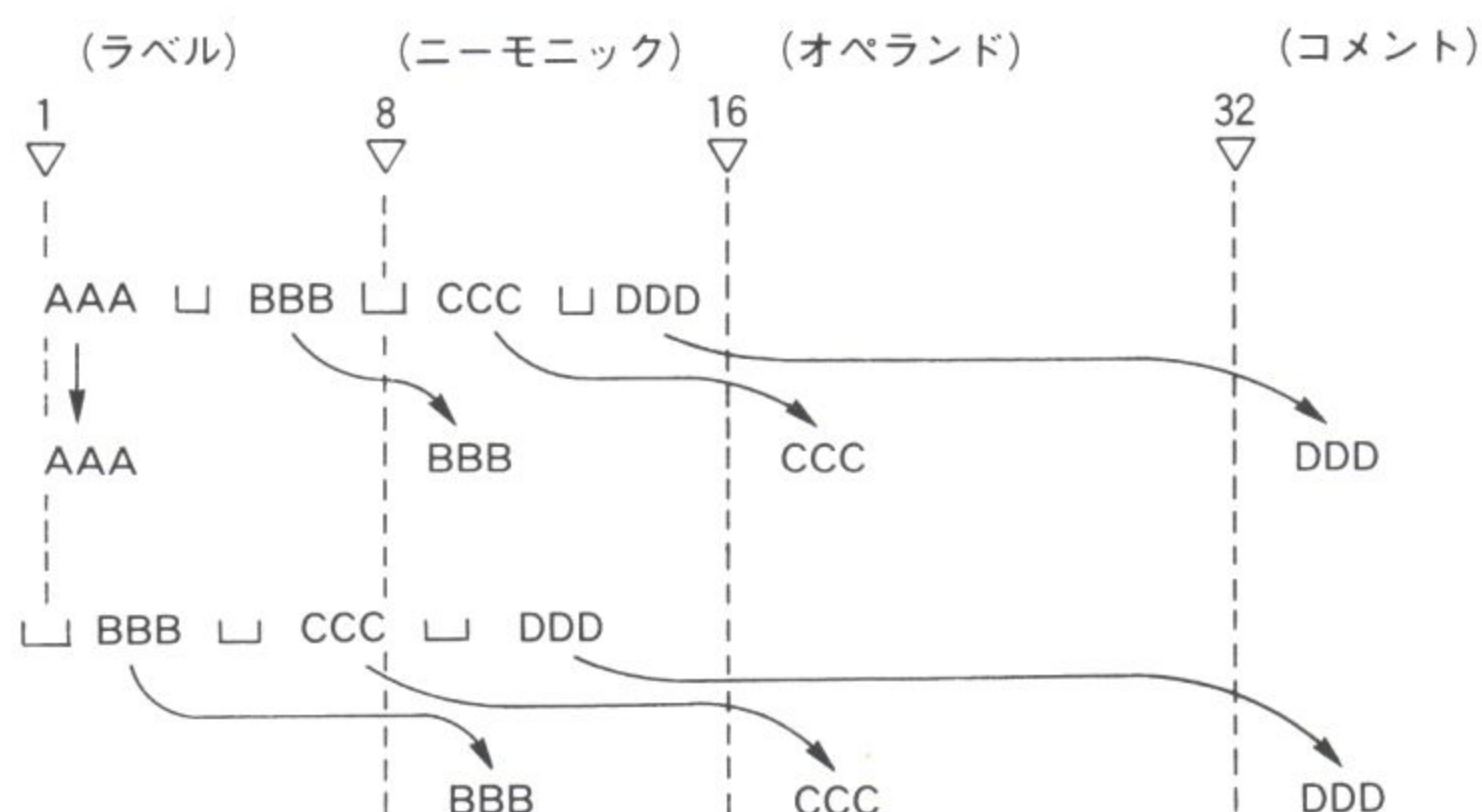
本章では，BASICによるユーティリティ・プログラムとして，タブ処理を行なうプログラムと，C変換，外部関数を新設する方法について紹介します．

このうち，外部関数についてはアセンブラの知識が必要なので，第5部の内容を理解した上で読んでいただくのが抵抗の少ない手順です．

C変換については，高速化やコマンド・パラメータの取り込みなどの魅力的なメリットがあり，本章でもこの点を活用したプログラムのサンプルを掲載しています．

●図5．2　アセンブラではソースの内容が位置合わせされないままリストに出力される

```
  1 00000000                          *
  2 00000000                          *      None Macro
  3 00000000                          *
  4 00000000
  5 00000000                           include ¥include¥doscall.mac
  5 00000000                                  .list
  6 00000000
  7 00000000                          start:
  8 00000000 4879(01)0000000C          pea text_line *Set text address
  9 00000006 FF09                      dc.w   _PRINT *DOS-call
 10 00000008 588F                      addq.l #4,sp *Release stack
 11 0000000A FF00                      dc.w _EXIT *End of program
 12 0000000C                          *
 13 0000000C 54657374206F6B0D         text_line dc.b 'Test ok',$0d,$0a,0
             0A00
 14 00000016                           end start
```

●図5．3　アセンブラ・ソース・プログラムのタブ処理の概要

第1字がスペースのときはラベルなしとみなし，ニーモニックから配置する．

●図5．4　タブ編集を行なう BASIC プログラム（入出力ファイル名は実行を開始してからキーボードより入力する）

```
 10 /* Auto. TAB Editor */
 20 int fn,fno1,fno2
 30 dim char cr(2)={&HD,&HA}
 40 str infile[20]
 50 str outfile[20]
 60 str fistr[120]
 70 str fostr[120]
 80 input " Input File = ",infile
 90 input "Output File = ",outfile
100 fno1=fopen(infile,"r")
110 fno2=fopen(outfile,"c")
120 while feof(fno1)<>-1
130      freads(fistr,fno1)
140      tab_set()
150      fwrites(fostr,fno2)
160      fwrite(cr,2,fno2)
170 endwhile
180 fclose(fno1)
190 fclose(fno2)
200 end
210 /* Line Editor */
220 func tab_set()
230   int x=1,y
240   if left$(fistr,1)="*" then {
250        fostr=fistr
260      } else {
270        fostr=""
280        tab_pnt=1
290        if mid$(fistr,x,1)<>" " then x=tab_sub(x) else {
300             x=tab_scan(x) }
310        tab_pnt=8
320        x=tab_sub(x)
330        tab_pnt=16
340        x=tab_sub(x)
360        tab_pnt=32
370        x=tab_sub(x)
380   }
390 endfunc
400 /* Shift to TAB */
410 func tab_sub(x)
420   int y
430   if x<len(fistr)+1 then {
440        if tab_pnt<>1 then {
450             if len(fostr)>=tab_pnt then {
460                  fostr=fostr+" "
470                } else {
480                  while len(fostr)<tab_pnt
490                       fostr=fostr+" "
500                  endwhile
510             }
520        }
530        if tab_pnt=32 then y=len(fistr)+1 else {
540             if mid$(fistr,x,1)="'" then {
550                  y=instr(x+1,fistr,"'")
560                } else {
565                  y=x
566             }
570             y=instr(y,fistr," ")
590             if y=0 then y=len(fistr)+1
600        }
610        fostr=fostr+mid$(fistr,x,y-x)
620        y=tab_scan(y)
630   } else y=x
640 return(y)
650 endfunc
660 /* Scan to Next Block */
670 func tab_scan(x)
680   while mid$(fistr,x,1)=" "
690        x=x+1
700   endwhile
710 return(x)
720 endfunc
```

●図5．5　タブ処理を行なったソースによりアセンブルしたリスト

```
     1 00000000                        *
     2 00000000                        *      None Macro
     3 00000000                        *
     4 00000000
     5 00000000                                include ¥include¥doscall.mac
     5 00000000                                .list
     6 00000000
     7 00000000                        start:
     8 00000000 4879(01)0000000C               pea      text_line        *Set text addres
s
     9 00000006 FF09                           dc.w     _PRINT           *DOS-call
    10 00000008 588F                           addq.l   #4,sp            *Release stack
    11 0000000A FF00                           dc.w     _EXIT            *End of program
    12 0000000C                        *
    13 0000000C 54657374206F6B0D       text_line dc.b   'Test ok',$0d,$0a,0
                0A00
    14 00000016                                end      start
```

第1字がスペースのものはラベルに相当するブロックがないものとして扱います．このとき，1つのブロックの内容が次のタブ位置にかかるほど長くなる場合は，1つだけスペースを置いて次のブロックに続けます．また，第1字が“＊”の場合はコメント行なので，タブ処理を行なうとかえって見苦しくなることがあります．したがって，コメントについては，タブ処理をバイパスするようにしなければなりません．

このような編集は，図5.4のプログラムで処理できます．ここではタブ処理前の入力ファイル名と，処理結果を収容する出力ファイル名とを input 命令によってキーボードから入力し，指定されたファイル間で処理を行なうようになっています．

プログラムはCライクに書かれ，これによって処理したファイルを使ってアセンブルした結果は，図5.5のように整然としたものになりました．このプログラムでは，文字定数(“＇”で始まり“＇”で終わるもので，間にスペースを含む場合がある)についての配慮もしてあり，オペランドの後のコメントに含まれるスペースもそのまま残すようになっています．

5-2 C変換するプログラムの注意点

C変換を前提とするBASICプログラムには，変換上の都合から多少の制限があります．

たとえば，Cではプログラムはmain関数を中心にもっているので，この名前を関数名に使用すると衝突してしまいます．また変換プログラム(BASTOC；実行時コマンド名“BC”)で生成する，goto，gosub命令の飛び先との衝突も考慮しておかなければなりません．すなわち，goto命令については，“L”，gosub命令では“S”の後に行番号が付いたラベルが作られるため，同名のラベルを定義しないようにします．同様に，bastocは“b_”で始まるラベルも生成するので，注意しなければなりません．ユーザ定義の関数名で，“$”で終わっているものは，“S”に置換されるという点についても承知しておく必要があります．

一般に式の記述は，CとBASICの文法上の違いもあって，演算順序について意識してかかる必要があります．どちらも矛盾なく処理するには，カッコを利用するのがベストです．また，論理演算結果の値が真の場合は0ですが，偽の場合BASICでは−1，Cでは1となります．結果を数値として利用するときには，とくに偽の場合注意が必要です．

Cの関数構造からいって，メイン部の一部がサブルーチンとなる関数の一部やその後にgotoするようなことは許されません．同様に，関数の内部から外部にgotoすることも不適当です．また，funcで定義したサブルーチンは，手続きの異なるgosub命令で呼び出すことはできません．gosub型のサブルーチンは，リカーシブ・コールができないなど制約が多いので，使わないほうがよいでしょう．

メイン部から参照される関数では，データはローカル変数として宣言されます．具体的にはスタックを利用して処理することになるので，大きな配列を定義すると，はみ出す心配があります．このような場合

は，メイン部で定義すべきです．

また，BASICプログラムでは，Cライブラリで用意されている関数も利用できますが，strlwr，strupr，strrevについては，引数として直接文字列定数を与えることができません．

その他の注意点としては，key　list命令に対応するCの関数が用意されていないので，これらの命令は使えません．もっとも，BASICプログラム中でこれらの命令を使用するケースもあまりないはずです．

BASICの文法との関連では，print命令のパラメータの区切りとして許されているのは，“,”と“;”だけなので，BASICでたまたまスペースが使えるからといって，BASTOCで正しく処理されるということは期待できません．また，配列の宣言時には“=”を使って初期値の設定を一度にできますが，それ以外で同じことを要求してもうまくいきません．

以上の点に注意すれば，BASTOCで変換可能なプログラムを作成できます．

なお，Cに変換するプログラムの拡張子は，“.b”または“.bas”とします．これは，Cコンパイラ・ドライバ(cc)にBASICプログラムからの処理であることを知らせるためです．

5-3　C変換してコマンドのパラメータを取り込む

X-BASICでは，Cに変換することによって実行速度が改善されるだけでなく，コマンド・ラインの内容を読み取り，パラメータを取得することができます．このことが操作性に寄与するところは大きく，一般のプログラムのように，コマンド・ラインだけでそのプログラムへの指示が行なえるようになります．これによって，起動するコマンドをバッチ・ファイルに登録しておいて，バッチ・ファイル名だけで他のプログラム(コマンド)と一緒に実行することが可能になります．

コマンド・ラインの内容は，図5.6のように，**b_argv**という配列に，**b_argc**の値が示すバイト数の長さで与えられます．このとき，コマンドは最終的なパス・リストに変換されており，その後にパラメータが続いています．ここで，コマンドのパス・リストとパラメータの間には目印となるスペースなどのデータが入っていないので，境界線をプログラムで見つけ出す工夫をしなければなりません．コマンドのパス・リストは，そのプログラムを収容するディレクトリ名などの長さによって変化するので，常に固定長であるとは限らないためです．

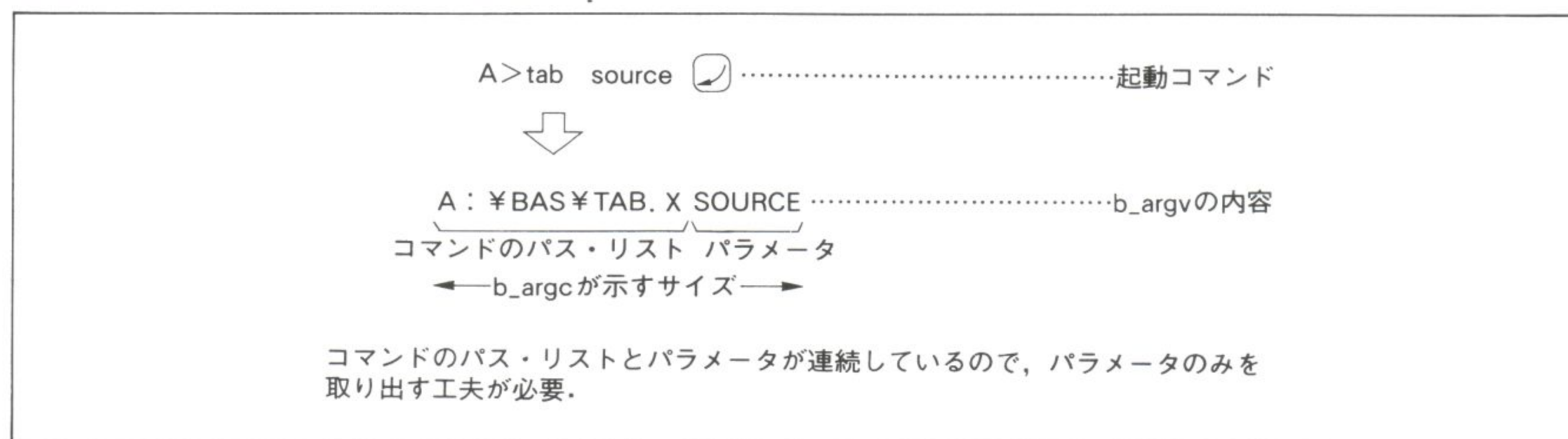
ここではTAB編集プログラムをアレンジして，ファイル名をパラメータで与える方法にすることを考えます．このとき入力と出力のファイル名を共通にし，拡張子を入力ファイルでは，“.dat”，出力ファイルでは“.s”とします．こうしておいて，共通部だけ入力するのです．

プログラムは図5.7のとおりで，メイン部分を，パラメータによって入出力ファイル名が生成できるように変更しただけです．

C変換と同時に実行形式の機械語ファイル作成まで進めるには，

```
cc <ソース・ファイル名>
```

●図5．6　コマンドと参照時のb_argv，b_argcの関係

```
A>tab  source ⏎ ·································起動コマンド
        ⇩
A：¥BAS¥TAB.X SOURCE ··························b_argvの内容
コマンドのパス・リスト  パラメータ
←―b_argcが示すサイズ―→

コマンドのパス・リストとパラメータが連続しているので，パラメータのみを
取り出す工夫が必要．
```

●図５．７　タブ処理プログラムにコマンド・パラメータ取得機能を加えたもの

```
 10 /* Auto. TAB Editor CC-Version */
 20 int fn,fno1,fno2,csiz,i
 30 dim char cr(2)={&HD,&HA}
 40 str parbuf[40]
 50 str infile[20]
 60 str outfile[20]
 70 str fistr[120]
 80 str fostr[120]
 90 csiz=b_argc
100 for i=0 to csiz-1
110     parbuf=parbuf+b_argv(i)
120 next
130 infile=right$(parbuf,len(parbuf)-instr(1,parbuf,"tab.x")-4)
140 outfile=infile+".s"
150 infile=infile+".dat"
160 print infile+" to "+outfile
170 fno1=fopen(infile,"r")
180 fno2=fopen(outfile,"c")
190 while feof(fno1)<>-1
200     freads(fistr,fno1)
210     tab_set()
220     fwrites(fostr,fno2)
230     fwrite(cr,2,fno2)
240 endwhile
250 fclose(fno1)
260 fclose(fno2)
270 end
280 /* Line Editor */
290 func tab_set()
300   int x=1,y
310   if left$(fistr,1)="*" then {
320       fostr=fistr
330     } else {
340       fostr=""
350       tab_pnt=1
360       if mid$(fistr,x,1)<>" " then x=tab_sub(x) else {
370           x=tab_scan(x) }
380       tab_pnt=8
390       x=tab_sub(x)
400       tab_pnt=16
410       x=tab_sub(x)
420       tab_pnt=32
430       x=tab_sub(x)
440   }
450 endfunc
460 /* Shift to TAB */
470 func tab_sub(x)
480   int y
490   if x<len(fistr)+1 then {
500       if tab_pnt<>1 then {
510           if len(fostr)>=tab_pnt then {
520               fostr=fostr+" "
530             } else {
540               while len(fostr)<tab_pnt
550                   fostr=fostr+" "
560               endwhile
570           }
580       }
590       if tab_pnt=32 then y=len(fistr)+1 else {
600           if mid$(fistr,x,1)="'" then {
610               y=instr(x+1,fistr,"'")
620             } else {
630               y=x
640           }
650           y=instr(y,fistr," ")
660           if y=0 then y=len(fistr)+1
670       }
680       fostr=fostr+mid$(fistr,x,y-x)
690       y=tab_scan(y)
700   } else y=x
710 return(y)
720 endfunc
```

```
730 /* Scan to Next Block */
740 func tab_scan(x)
750   while mid$(fistr,x,1)=" "
760       x=x+1
770   endwhile
780 return(x)
790 endfunc
```

コマンドを使います．これによってBASTOC，Cコンパイラ，アセンブラ，リンカが次々と自動的に働きます．なおccの詳細については，第4部で説明します．

5-4 BASIC外部関数を作るには

この節の説明は，第5部(アセンブラ)が理解できてから読むとわかりやすいでしょう．

BASICの外部関数は，単に使える関数を増やすという目的だけでなく，ユーザ作成の機械語(アセンブラ)プログラムを連結して実行させるという意味をももっています．

外部関数を組み込むためには，".FNC"という拡張子の付いたプログラム・ファイルを作成して，"BASIC"のあるディレクトリ内に置き，その名前を同ディレクトリ内の"BASIC，CNF"に登録します．登録に際してはエディタを使い，

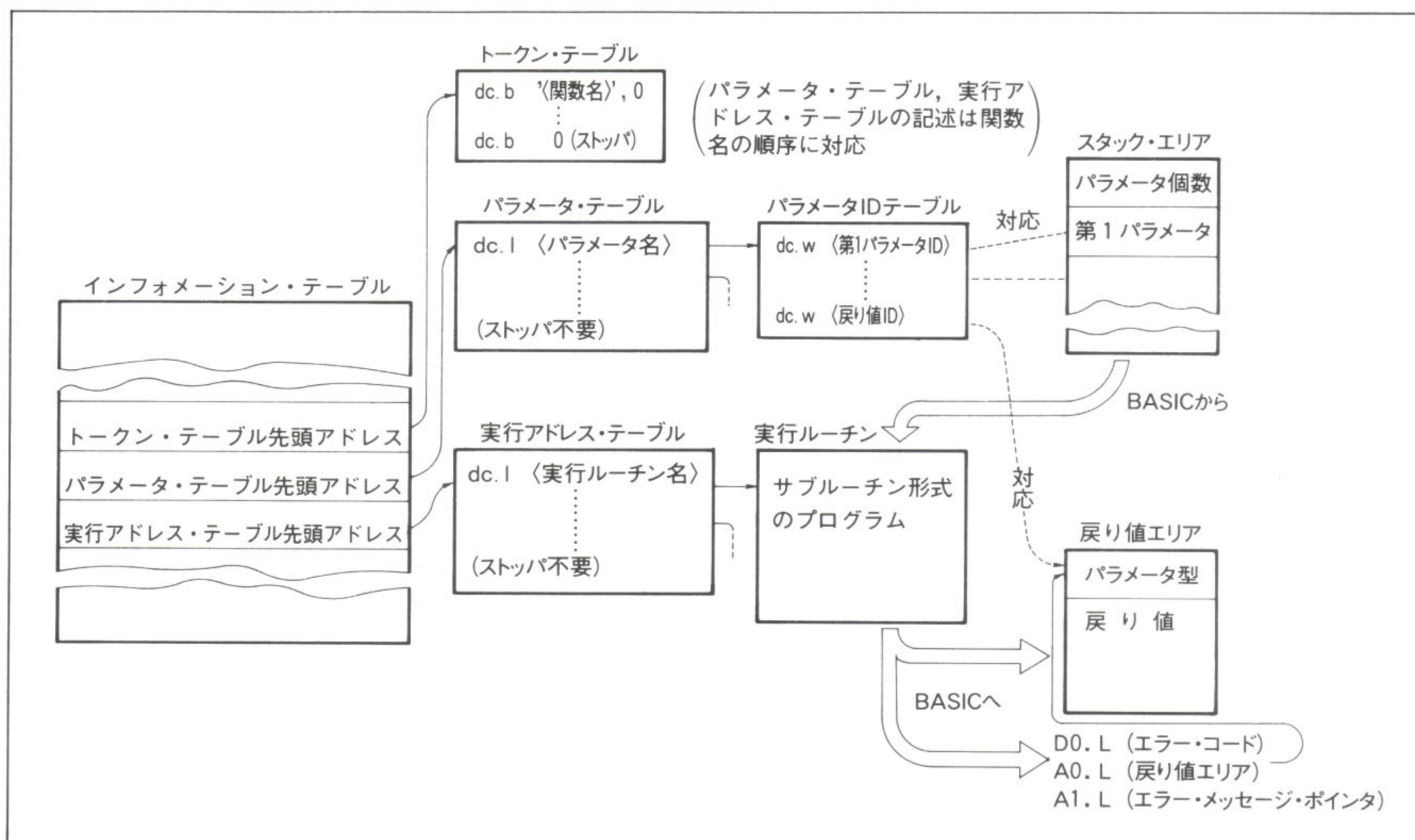
FUNC ＝ 〈外部関数ファイル名〉

のように追記するだけです．

問題の外部関数プログラム・ファイルの中で最大の関心事は，実行時のサブルーチンをどう作るかということですが，それ以外にもたくさんの前準備を必要とします．図5.8はそれらの関連を一覧表にしたもので，たった1つの外部関数を新設するにもこれだけの素材を準備しなければなりません．

このうち**インフォメーション・テーブル**はBASIC実行の各段階でユーザ・ルーチンへの入口が設定できるよう用意されているもので，必ずプログラムの先頭に置かれます．**トークン・テーブル**は外部関数名を

●図5．8　BASIC外部関数のテーブルなどの関連図

トークン・テーブル
dc.b '〈関数名〉', 0
dc.b 0 (ストッパ)
(パラメータ・テーブル，実行アドレス・テーブルの記述は関数名の順序に対応)
スタック・エリア
パラメータ個数
第1パラメータ
パラメータ・テーブル
dc.l 〈パラメータ名〉
(ストッパ不要)
パラメータIDテーブル
dc.w 〈第1パラメータID〉
dc.w 〈戻り値ID〉
対応
インフォメーション・テーブル
トークン・テーブル先頭アドレス
パラメータ・テーブル先頭アドレス
実行アドレス・テーブル先頭アドレス
BASICから
実行アドレス・テーブル
dc.l 〈実行ルーチン名〉
(ストッパ不要)
実行ルーチン
サブルーチン形式のプログラム
対応
戻り値エリア
パラメータ型
戻り値
BASICへ
D0. L (エラー・コード)
A0. L (戻り値エリア)
A1. L (エラー・メッセージ・ポインタ)

収容し，BASICが目的の関数を検索するのに用いられます．**パラメータ・テーブル**，**実行アドレス・テーブル**は，その順序に従ってそれぞれパラメータIDテーブルのアドレス，実行アドレスを配置します．**パラメータIDテーブル**は各関数ごとに必要なもので，パラメータの順序に従って，そのデータ型を表記するIDを登録します．

5-5 外部関数関連テーブルの作り方

●インフォメーション・テーブル

このテーブルには，ロング・ワードで表5.1に示すアドレス値が列記されます．ただし，このうち外部関数で通常使用するのは，アミがけ部分のみで，1～8番はダミーとするのが普通です．BASICでは，表に示すタイミングで一応ユーザ・ルーチンに飛ばすようになっているので，ダミーの場合はRTS命令で何もせずに終了させます．

トークン・テーブル，パラメータ・テーブル，実行アドレス・テーブルの先頭アドレス値は，各テーブルに付けたラベル名を，

```
dc.l 〈ラベル名〉
```

形式で記述することに尽きます．ダミーとする項目は，RTS命令に付けたラベル名を記載すればよいのです．

●トークン・テーブル

このテーブルは，関数名を参照し，何番目に登録されているかを調べるのに使われます．したがって，目的の関数が登録されていないときは最後まで参照されるので，ストッパとして1バイトの0が必要です．また，関数名は64字までの可変長となっているので，そのためのストッパとして個別に1バイトの0を末尾に付けます．そこで，1つの関数名は，

```
dc.b '〈関数名〉', 0
```

のように書き，最後の関数名の次に，

```
dc.b 0  (全体のストッパ)
```

を置いて締めくくります．

●表5.1 インフォメーション・テーブルの内容

記載順	アドレスのオフセット値	サイズ（バイト）	内容
1	+0	4	BASIC起動時の初期化ルーチンのアドレス
2	+4	4	RUN時に実行されるサブルーチンのアドレス
3	+8	4	END時に実行されるサブルーチンのアドレス
4	+12	4	SYSTEMコマンド実行時やEXIT命令によるOS復帰時に実行されるサブルーチンのアドレス
5	+16	4	BREAK の実行時や，CTRL+C によるプログラムの中断時に呼ばれるサブルーチンのアドレス
6	+20	4	一行入力中の CTRL+D の押下時に呼ばれるサブルーチンのアドレス
7	+24	4	（予備）
8	+28	4	（予備）
9	+32	4	トークンテーブルの先頭アドレス
10	+36	4	パラメータテーブルの先頭アドレス
11	+40	4	実行アドレステーブルの先頭アドレス
12	+44～63	20	（予備）0を入れておく

●表5．2　パラメータIDの値と意味

パラメータ

パラメータID	意　　　味	fdef. hによる名前
$0001	8バイト浮動小数点型実数	float_val
$0002	4バイト符号付き整数	int_val
$0004	1バイト符号なし整数	char_val
$0008	文字列	str_val
$0011	浮動小数点型のデータ部のポインタ	float_vp
$0012	int型変数のデータ部のポインタ	int_vp
$0014	char型変数のデータ部のポインタ	char_vp
$0018	文字列型変数のデータ部のポインタ	str_vp
$0081	省略可能な8バイト浮動小数点型実数	float_omt
$0082	省略可能な4バイト符号付き整数	int_omt
$0084	省略可能な1バイト符号なし整数	char_omt
$0088	省略可能な文字列	str_omt

戻り値

パラメータID	意　　　味	fdef. hによる名前
$8000	8バイト浮動小数点型実数	float_ret
$8001	4バイト符号付き整数	int_ret
$8003	文字列	str_ret
$FFFF	戻り値なし	void_ret

※char型は戻り値としては存在しない．

●パラメータ・テーブル

このテーブルは，トークン・テーブルで目的の関数が見つかった場合，その関数のパラメータ構造の記述場所を知るのに用いられます．したがって，トークン・テーブルの記述順に，

dc.l　<パラメータ　IDテーブル名>

形式で，パラメータIDテーブルのアドレス定数を並べればよいのです．このとき，該当関数の出現位置はすでにわかっているのでストッパはいりません．

●実行アドレス・テーブル

BASICで外部関数名の使われる命令が実行されたときに，対応するサブルーチンの入口にあたるラベル名を，

dc.l　<実行ルーチン名>

形式で，トークン・テーブルの順序に並べます．このテーブルもパラメータ・テーブルと同様な手法で参照されるので，ストッパは不要です．

●パラメータIDテーブル

1つの外部関数について，そのパラメータ構造を記述するテーブルです．したがって，このテーブルは，トークン・テーブルに定義されている外部関数の数だけ作られることになります．

テーブルの内容は，第1パラメータから順に戻り値までのデータ型(パラメータID：表5.2)を並べればよいのですが，パラメータと戻り値とでは同じデータ型でもID値が異なっている点に注意が必要です．これは，戻り値の記述の最上位ビットをストッパとしても兼用しているためです．また，戻り値として1バイトのキャラクタが使用できない点も知っておいたほうがよいでしょう．

表中の“int_val”などの表意定数は，

```
¥include ¥fdef.h
```

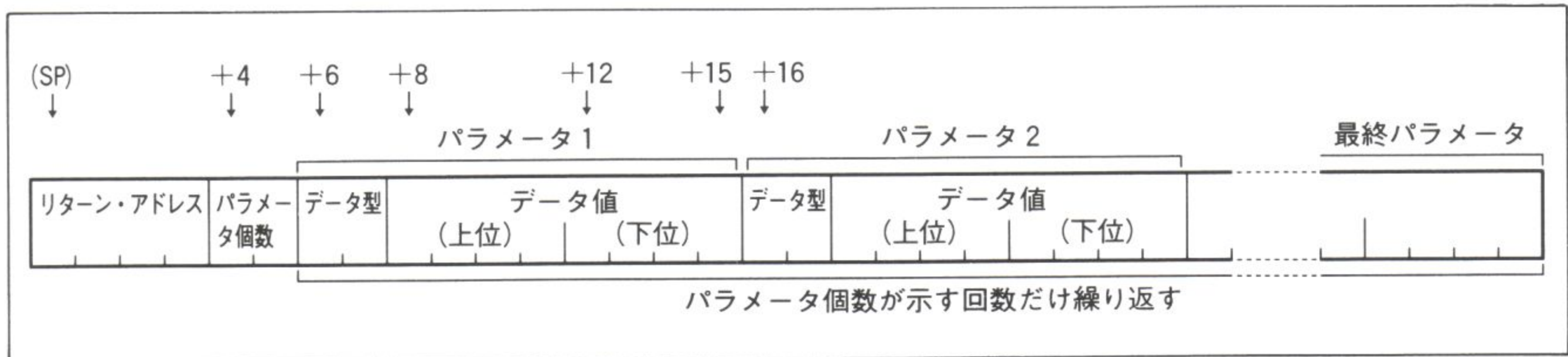
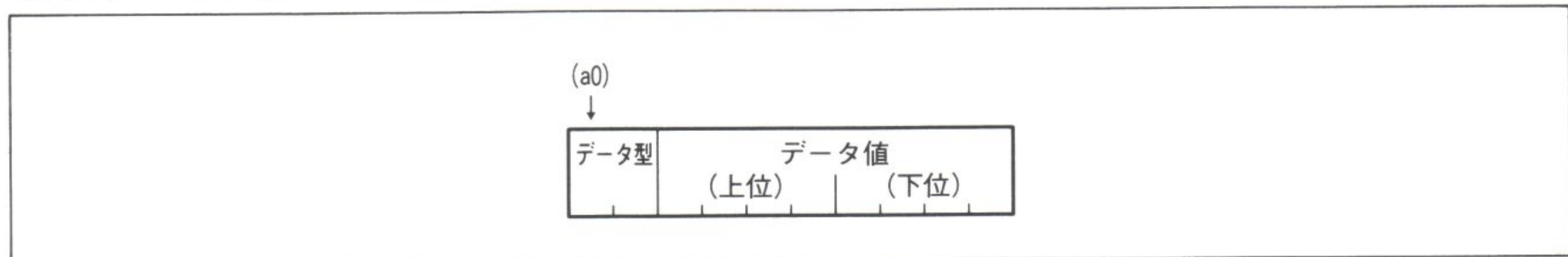
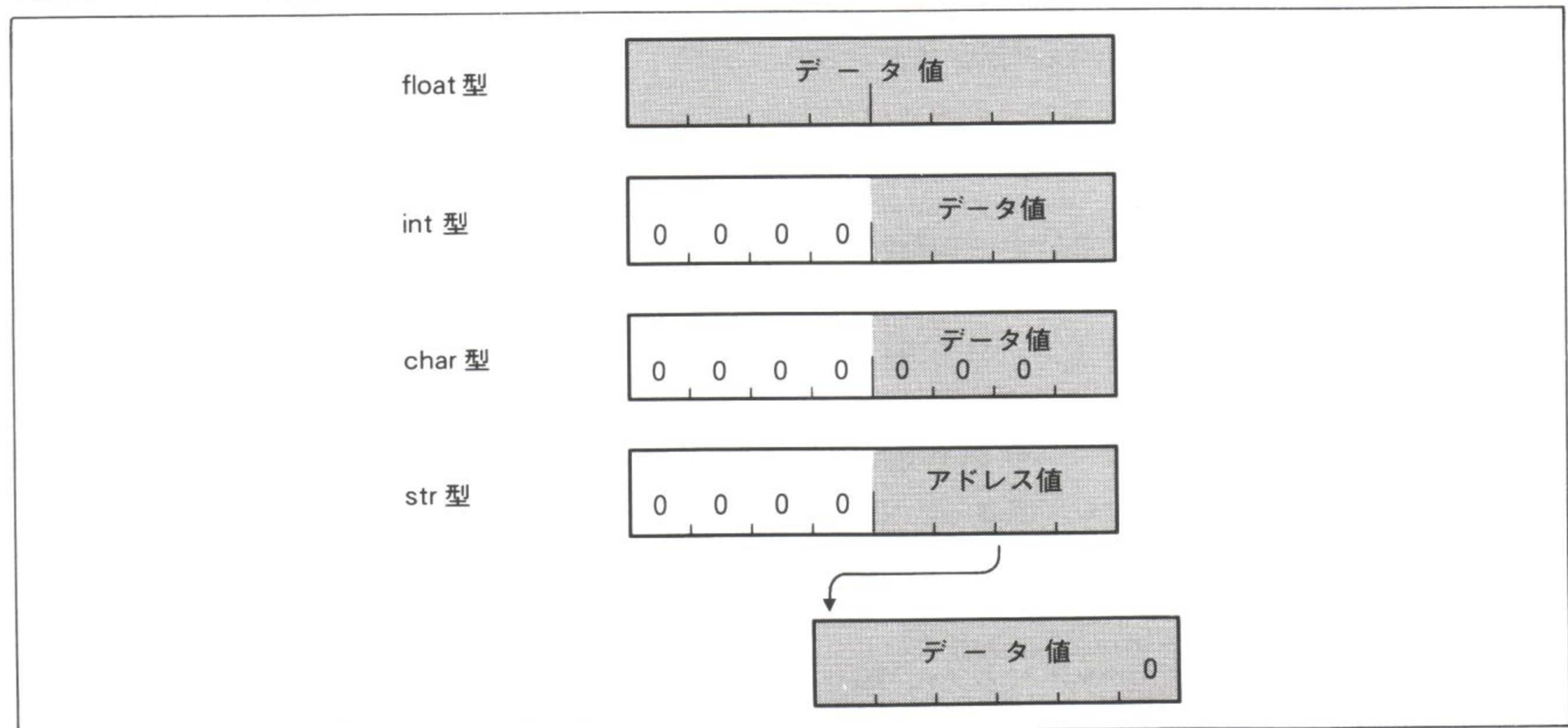
にequ定義が登録されているので，パラメータid値を直接書くよりも，このファイルをinclude疑似命令で参照して表意定数記述とするほうが，プログラムのわかりやすさの点でベターです．

なお，各表に記述されるアドレス値(実際はラベル名)は偶数値でなければなりません．と言うことは，定義先では偶数アドレスから始まっている必要があります．このことは，定義内容がdc.lやdc.w，あるいはプログラム命令ならば当然守られるべき事柄でもあります．

5-6 実行ルーチンで参照するデータについて

個別の関数に対応する実行ルーチンは，スタックに置かれたBASICからのパラメータ(図5.9)を受け取り，処理をしたあと，戻り値(図5.10)をBASICに引き渡しする形で作成されます．

パラメータは関数の引数に対応するもので，１つのパラメータは10バイト占有します．その内訳はデータ型２バイト，データ値８バイトとなっていますが，データ値部分の使われ方はデータ型によって異なります(図5.11)．

●図５．９　スタックのデータ構造（BASIC→）

●図５．10　戻り値のデータ構造（→BASIC）

●図５．11　データ型によるデータ値の入り方の違い

●表5．3　データ型コードとその意味

型を示す値	パ ラ メ ー タ の 型
0	8バイトの浮動小数点型実数（float 型）
1	4バイトの符号付き整数（int 型）
2	1バイトの符号なし整数（char 型）
3	文字列型（str 型）
$FFFF	省略された引数（void 型）

たとえば，パラメータ1のデータ値はfloat型ならばsp＋8から，int型なら＋12から，char型なら＋15の位置に実際の値が入ります．唯一の例外はstr型で，データ値の入る位置の下位4バイトに，実際の文字列が収容されている領域のアドレス値が入っています．

データ型を区別するコードは，表5.3のとおりで，パラメータIDと異なる部分があるので注意が必要です．

プログラムでは，内部のサブルーチンのリターン・アドレス用などでスタック・ポインタ(SP)の値を変更できますが，戻るときには復元されていなければなりません．これは，BASICから渡されるスタックの先頭にBASICへのリターン・アドレスが収容されているからです．

実行ルーチンは，このリターン･アドレスを利用し，処理終了時にはRTS命令でBASICに帰る，いわばサブルーチン形式で作成されます．

なお，内部サブルーチンからパラメータを参照するときは，スタックの相対位置がズレているので注意が必要です．

5-7　外部関数新設の実例(peek，poke，exec)

ここでは，これまで説明した事柄の実践例として，マイクロソフト系BASICで使われているpeek,poke,exec命令に相当する外部関数を新設するケースについて紹介します．

peekは，指定されたアドレスから1バイトのデータを読み出し，関数の左辺(結果)に転送する命令で，これに対しpokeは指定されたアドレスに，指定された1バイト・データを書き込む命令です．peekは読み出しだけなので比較的安全ですが,pokeはメモリの内容を変更してしまうので注意して使わないと危険です．execはpokeなどによってメモリに書き込まれた機械語プログラムを実行させるもので，ゲーム・ソフトなどではよくこのコンビネーションを見かけます．

X-BASICに移植する際の文法は，それぞれ

```
  〈結果の変数〉＝peek(〈読み出しアドレス〉)
poke(〈書き込みアドレス〉，〈データ〉)
exec(〈アドレス〉)
```

の形式とし，変数は簡単にするためすべて整数型に限定します．

図5.12は，このためのアセンブラ・プログラムです．

peekの実行ルーチンでは，アドレス値を受け取って該当位置からデータを読み出し，戻り値に転送します．このとき1バイトのみなので，データ値は最下位に置きます

同様に，poke実行ルーチンでデータ値を受け取る際には，最下位から1バイトのみ取り出し，それを転送先で指定されたアドレスに対して書き込んでいます．

exec実行ルーチンは，単に指定アドレスのサブルーチンを実行させるだけです．

これらの新設関数をテストするために，図5.13の機械語プログラムをBASICからメモリに置き，実際に実行させてその結果をダンプするプログラム(図5.14)を用意しました．機械語のテスト・プログラムは，F0020番地に＄1200を書き，次に同じ位置に＄34を加算するものです．実行結果が＄1234になればOKです．

●図5．12　外部関数 peek, poke, exec を新設するためのプログラム

```
 1 00000000                            ******************************
 2 00000000                            *     BASIC User Function    *
 3 00000000                            ******************************
 4 00000000                                    include ¥include¥fdef.h
 4 00000000                                    list
 5 00000000                            *
 6 00000000                            *     Information Table
 7 00000000                            *
 8 00000000 (01)00000040                       dc.l    F_init
 9 00000004 (01)00000040                       dc.l    F_run
10 00000008 (01)00000040                       dc.l    F_end
11 0000000C (01)00000040                       dc.l    F_exit
12 00000010 (01)00000040                       dc.l    F_break
13 00000014 (01)00000040                       dc.l    F_ctrlD
14 00000018 (01)00000040                       dc.l    F_dmy1
15 0000001C (01)00000040                       dc.l    F_dmy2
16 00000020 (01)00000042                       dc.l    F_token
17 00000024 (01)00000052                       dc.l    F_parTbl
18 00000028 (01)0000006C                       dc.l    F_exec
19 0000002C 0000000000000000                   dc.l    0,0,0,0,0
            0000000000000000
            00000000
20 00000040
21 00000040                            *     Dummy
22 00000040                            F_init:
23 00000040                            F_run:
24 00000040                            F_end:
25 00000040                            F_exit:
26 00000040                            F_break:
27 00000040                            F_ctrlD:
28 00000040                            F_dmy1:
29 00000040                            F_dmy2:
30 00000040 4E75                               rts
31 00000042
32 00000042                            *
33 00000042                            *     Token Table
34 00000042                            *
35 00000042                            F_token:
36 00000042 7065656B00                         dc.b    'peek',0
37 00000047 706F6B6500                         dc.b    'poke',0
38 0000004C 6578656300                         dc.b    'exec',0
39 00000051 00                                 dc.b    0
40 00000052                                    even
41 00000052
42 00000052                            *
43 00000052                            *     Parameter Table
44 00000052                            *
45 00000052                            F_parTbl:
46 00000052 (01)0000005E                       dc.l    Peek_par
47 00000056 (01)00000062                       dc.l    Poke_par
48 0000005A (01)00000068                       dc.l    Exec_par
49 0000005E
50 0000005E                            *
51 0000005E                            *     Parmetr ID Table
52 0000005E                            *
53 0000005E                            Peek_par:
54 0000005E 0002                               dc.w    int_val
55 00000060 8001                               dc.w    int_ret
56 00000062
57 00000062                            Poke_par:
58 00000062 0002                               dc.w    int_val
59 00000064 0002                               dc.w    int_val
60 00000066 FFFF                               dc.w    void_ret
61 00000068
62 00000068                            Exec_par:
63 00000068 0002                               dc.w    int_val
64 0000006A FFFF                               dc.w    void_ret
65 0000006C
66 0000006C                            *
67 0000006C                            *     Execution Table
68 0000006C                            *
```

```
  69 0000006C                          F_exec:
  70 0000006C (01)00000082                     dc.l    Peek_func
  71 00000070 (01)00000096                     dc.l    Poke_func
  72 00000074 (01)000000A2                     dc.l    Exec_func
  73 00000078
  74 00000078                          *
  75 00000078                          *    PEEK Function
  76 00000078                          *
  77 00000078 0001                     ret_par dc.w    1              Int_val
  78 0000007A 00000000                         dc.l    0
  79 0000007E 000000                           dc.b    0,0,0
  80 00000081 00                       val     dc.b    0              Peeked Data
  81 00000082
  82 00000082                          Peek_func:
  83 00000082 206F000C                         move.l  12(sp),a0      Get Address
  84 00000086 13D0(01)00000081                 move.b  (a0),val       Read Data
  85 0000008C 4280                             clr.l   d0             Clear Error-stat
us
  86 0000008E 41F9(01)00000078                 lea     ret_par,a0     Set Response
  87 00000094 4E75                             rts
  88 00000096
  89 00000096                          *
  90 00000096                          *    POKE Function
  91 00000096                          *
  92 00000096
  93 00000096                          Poke_func:
  94 00000096 206F000C                         move.l  12(sp),a0      Get Address
  95 0000009A 10AF0019                         move.b  25(sp),(a0)    Write Data
  96 0000009E 4280                             clr.l   d0             Clear Error-stat
us
  97 000000A0 4E75                             rts
  98 000000A2
  99 000000A2                          *
 100 000000A2                          *    EXEC Function
 101 000000A2                          *
 102 000000A2
 103 000000A2                          Exec_func:
 104 000000A2 206F000C                         move.l  12(sp),a0      Get Address
 105 000000A6 4E90                             jsr     (a0)           Execution
 106 000000A8 4280                             clr.l   d0             Clear Error-stst
us
 107 000000AA 4E75                             rts
 108 000000AC
 109 000000AC                                  end
```

●図5．13　新設関数をテストするための機械語モデル・プログラム

```
  1 00000000                         *
  2 00000000                         *    Test Program of PEEK,POKE,and EXEC
  3 00000000                         *
  4 00000000 =000F0020               result  equ     $f0020
  5 00000000 33FC1200000F0020                move.w  #$1200,result
  6 00000008 06790034000F0020                addi.w  #$34,result
  7 00000010 4E75                            rts
  8 00000012
```

結果のダンプ(図5.15)を見ると，F0020番地から＄1234が入っているのが確認され，うまくいったことがわかります．

なお，もちろんpokeする領域については現在使用中でないことを確かめる必要がありますが，peekする領域についてもメモリのないところやシステム領域に使用するとバス・エラーとなります．また，書き換えができても他の目的で使われている領域では暴走する危険性があるので，最初にpokeやexecの行をコメントなどに変えてバイパスし，ダンプを確認してから正式に実行させるようにするのが適当です．

●図5．14　peek, poke, exec 関数をテストする BASIC プログラム

```
 10 /* Example of Using POKE,EXEC,and PEEK */
 20 int adr,dat,hcnt
 30 str hx
 40 /* Put Machine-language to Memory */
 50 dim char c(17)={&H33,&HFC,&H12,0,0,&HF,0,&H20,&H6,&H79,0,&H34,0,&HF,0,&H20
,&H4E,&H75}
 60 adr=&HF0000
 70 for x=0 to 17
 80   dat=c(x)
 90   poke(adr,dat)
100   adr=adr+1
110 next
120 /* Execution */
130 adr=&HF0000
140 exec(adr)
150 /* Memory Dump */
160 adr=&HF0000
170 for x=0 to &H2F
180   dat=peek(adr)
190   hx=hex$(dat)
200   if len(hx)=1 then hx="0"+hx
210   hcnt=hcnt+1
220   if hcnt=1 then print hex$(adr)+"   ";
230   print hx+" ";
240   if hcnt=16 then print:hcnt=0
250   adr=adr+1
260 next
270 end
```

●図5．15　テスト結果

```
F0000  33 FC 12 00 00 0F 00 20 06 79 00 34 00 0F 00 20
F0010  4E 75 00 00 00 00 00 00 00 00 00 00 00 00 00 00
F0020  12 34 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

第4部

Cプログラミング

第1章

Cの基礎知識

1-1 なぜCが流行するようになったか

「パソコン」という言葉がまだなく，「マイコン」で総称されていた時代の主流言語はアセンブラでした．

しかし，アセンブラを駆使するにはかなりの専門的知識が必要だというので，BASICが使えるようになってからは，すっかりBASICが中心になってしまいました．今日パソコンがこれほど使われるようになったのも，BASICの功績によるところが大きいといえます．

ただ，BASICには構造化プログラミングの考え方がなかったため，ソフトウェア開発の際に作業が乱雑になりがちで，エンジニアを育てるのに支障があるという批判が起きていたのも事実でした．このため，BASICの次に流行る言語は，BASICに似ていてかつ構造的なPASCALであろうという見方が有力だったのです．ところが，現実にはPASCALはあまり騒がれず，Cがその代わりに全面に出てしまいました．

CはUNIXのOSプログラムを開発するために考えられた言語で，「高級アセンブラ」と評されることもあります．つまり，実態は「関数言語」であり，サブルーチンを関数として使いながらプログラミングするのが特徴です．また，構造化プログラミングの考え方が反映されているので，ポストBASIC言語の資格をもしっかりともっています．

PASCALの欠点は，BASICと同様に細かいところはアセンブラで組んだサブルーチンでカバーしなければならないところにありました．Cの場合は，OSプログラム開発用として作られただけあって，かなりのところまではこなせるため，アセンブラに依存する度合いはかなり軽減しています．筆者が着目しているのはまさしくこの点で，BASICとはまったくイメージの異なるCがPASCALを押しのけて主役の座についたのは，ソフトウェア・エンジニアがBASICに飽きたというよりも，Cの場合これだけでほとんど用が足りるからなのです．

加えて，Cは記述が簡潔で，プログラムの入力が容易です．生産性を高める上でも役に立つとすれば，もてはやされるのも当然といえるでしょう．

第4部は，BASICはマスターしたがCは初めて，という読者に捧げます．

BASICに比べてCは難しいという声をよく耳にします．アセンブラに比べても，意外に簡単に見えるようで実はそうではないのがCの実態のようです．

そこで本章では，Cの基礎的な事柄を，できるだけイメージがとらえやすいように解説することにしました．Cの理解を妨げているものとしては，C特有の書き方にもその一因があり，同じ記号でも使う場所で意味が変わったりするものです．したがって読み進む際には常に，その記述がどういう意味をもっているかという点を意識してかかると理解が早いと思います．

説明の順序の都合から，ここでは，構造体など少し難しいデータ構造についても触れていますが，すぐにピンとこなければとりあえずパスし，後日またトライするほうがよいでしょう．

これまでA(アセンブラ)からB(BASIC)，Cと移り変わってきたので，今後“D”の付く言語が登場するかも知れません．

Cは68000と相性がよい!!

Cで書いたプログラムを，機械語に変換した(コンパイルした)結果のメモリ・サイズを比較すると，一般に8086系マシンよりも68000系のマシンのほうが圧倒的に小さくなります．これは68000 CPUがCに近い思想で設計されているため，変換効率がよいからです．

Cは実行効率のよいプログラミング言語といわれていますが，8086より5割以上も速いと言われる68000との組み合わせによって，さらにそのパワーを充分に発揮できることになります．

1-2 関数とCのプログラム構造

❖関連コマンド　　return

Cが関数言語であることはすでに述べたとおりですが,「関数」とは実質的にサブルーチンと同様であるということも第3部で触れました.

ここで，Cのプログラム構造の概念を図1.1に示します.

図では，プログラムには必ずメイン部(main)があって，サブルーチンに対応する関数はその下で動作することを示しています．また，main も含めて，関数の内容は“{”に始まり，“}”に終わる構造となっており，関数名の定義から“}”までを1つの区切りとみなせます.

関数は，別な関数を子のサブルーチンとして呼び出すことが可能です．そして，これらによって，サブルーチンの階層構造が形成されます.

Cでは，プログラムを終了させる return 文をとくに使わなくても，main の“}”までたどりついた時点がプログラム終了となります．ただし，戻り値が必要な場合は

　　return 〈式〉;

でその値をセットしなければなりません.

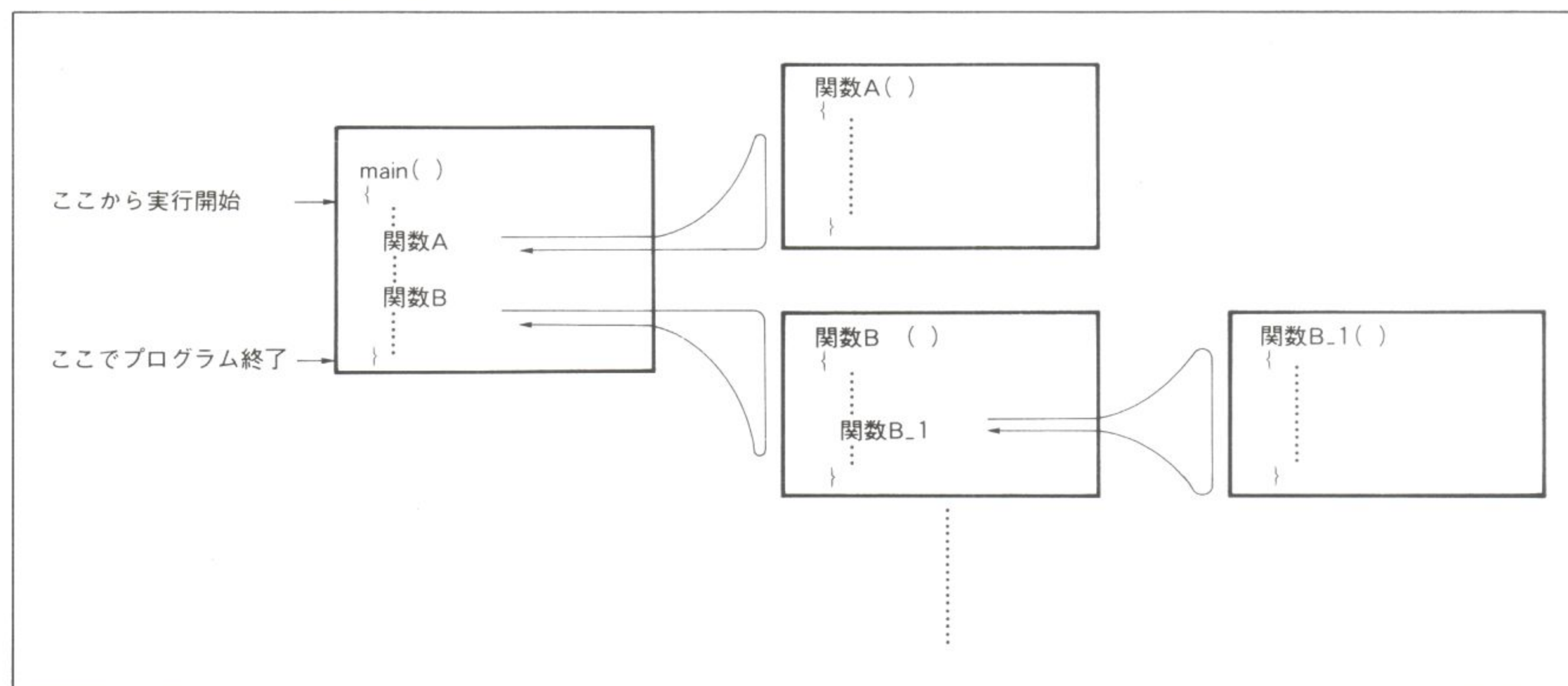
●図1．1　Cプログラムの構造

1-3 記憶クラスと変数の有効範囲

図1.3は各記憶クラスと変数の有効範囲について,プログラム(ソフトウェア)とハードウェアの関係をまとめたものです.

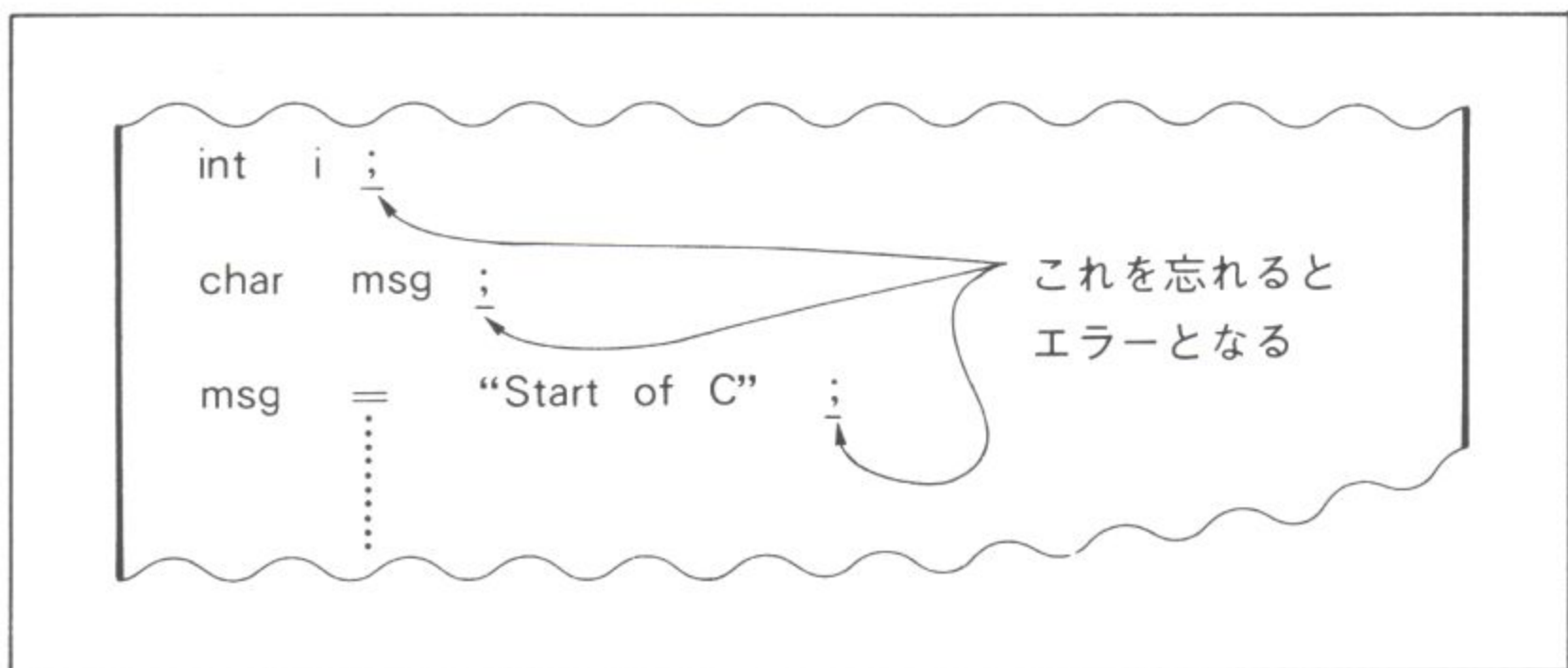
変数は大別して,CPU のレジスタ内にとられるもの(register),グローバル変数領域にとられるもの(extern, static)，ローカル変数領域にとられるもの(auto)に分類できます.

register(レジスタ)領域の変数は，CPU 内部にある利点を活かしてとくに高速にアクセスする必要のあるものに割り当てます．レジスタ数には限りがあり，システムが使用している以外のものしか使えないので，数個しか空きがありません．そこで足りないときは auto と同じ扱いに変更されます.

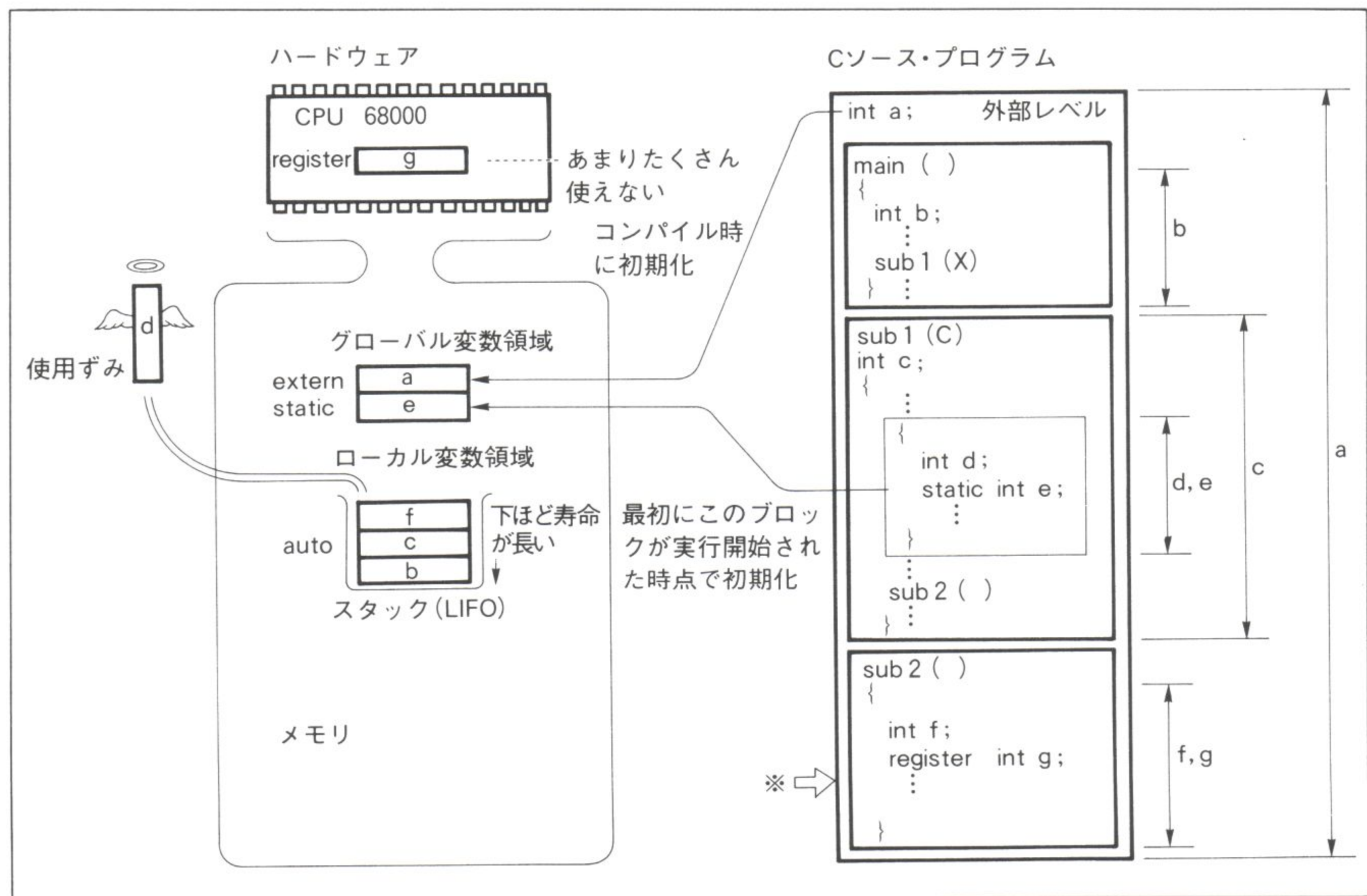
extern(外部)領域の変数は，プログラム全体から参照できます．同じく**グローバル変数領域**にありなが

Cでは“;”に注意

BASICなどと違って，Cでは命令の区切りに“;”を使っています．(図1.2)．もしこれが書かれていないと，次の行まで続いているとみなされ，たいていエラーになります．それも，エラー表示の行番号は次の行のほうになっているので，すぐには気付かないことが意外に多いものです．意味不明のエラーが出たら，1行上を疑ってみるとよいでしょう．

●図1．2　Cの命令記述の原則

ら，**static**(静的)領域の変数は定義されているブロック内でしか参照できません．しかし，グローバル変数領域の性質上，プログラム終了までデータ値は保存されます．

auto(自動)領域の変数については，**ローカル変数**としてメモリの**スタック領域**が使われます．この領域では，その関数またはブロックが実行を開始すると内部で定義された変数用の領域が取得され，終了とともに開放されます．その結果，あとで取得したものほど先に開放される(LIFO:ラスト・イン・ファースト・アウト)ことになり，空いたところは次のブロックが取得する際に利用されます．この領域では頻繁に取得，

●図1．3　＊点での各変数の存在場所と有効範囲

開放が行なわれるため，指定がなければ，初期化は行なわれません．

グローバル変数は固定された領域にあるので，extern 領域ではコンパイル時に初期値が詰められ(オブジェクト内に入る)，static 領域ではその変数が宣言されているブロックが最初に実行されたときに初期設定が行なわれます．初期値の指定がないときは，0でクリアされます．

static 領域がグローバル変数領域にあるにもかかわらず，その関数またはブロック内でしか参照できないのは，定義されたブロックと密着しているからです．むしろこの領域での存在理由は，下位のブロックで auto しか利用できないとすれば，次回へのデータ値の引き継ぎが必要なとき extern 領域を利用しなければならなくなるため，外部レベルの定義を余儀なくされるからという点にあります．下位のブロックのための定義は，あくまで下位に置くことが，プログラムをわかりやすくするために必要です．

なお，外部レベルでの領域の省略は extern に，内部レベル(各ブロック内)での省略は auto として扱われます．また内部レベルでの extern 指定は，外部定義を参照するという宣言であって，変数を新設するものではありません．

注意が必要なのは，関数内部で変数を定義する位置です．すなわち関数のパラメータに関するものは，ブロック("{" から "}" まで)の前に置き，関数内での作業領域的なものは，ブロック内に書くという規則になっているので，これに従わなければなりません．もしパラメータの定義をブロック内に書くと有効範囲の関係でパラメータと連動できなくなってしまいます．

1-4　データ型と変数の宣言

Cで扱うことができるデータ型の一覧を表1.1に示します．この中でもよく使われるのは int 型で，68000 CPU のレジスタ・サイズとも適合しているため，効率が非常によいというメリットがあります．

変数は次の型式で宣言されます．

[〈記憶クラス〉]　**〈型〉〈変数名〉**{,〈変数名〉}；

たとえば

```
int    a;
```

のように書き，記憶クラスは前節で述べた規則を利用して，とくに必要がない限り省略するのが普通です．また，変数名は同一記憶クラス，同一型のものを "," で区切って列記することができます．

変数に**初期値**を指定したいときは，

〈変数名〉＝ **〈定数式〉**

の形式で追記します．

●表1．1　Cのデータ型一覧

分類	型	サイズ	表現できる値の範囲
整数	char	8ビット	−128〜127
	int	32ビット	−2,147,483,648〜2,147,483,647
	short int	16ビット	−32,768〜32,767
	long int	32ビット	−2,147,483,648〜2,147,483,647
	unsigned char	8ビット	0〜255
	unsigned int	32ビット	0〜4,294,967,295
	unsigned short int	16ビット	0〜65,535
	unsigned long int	32ビット	0〜4,294,967,295
浮動小数点	float	32ビット	10^{-37}〜10^{38}
	double	64ビット	約10^{-307}〜10^{308}

このほかに enum 型，void 型がある．

戻り値のデータ型はなるべくintに

戻り値のデータ型は，その関数定義の先頭で行ないます．たとえばchar型ならば，関数名が“abc”のとき

```
char   abc(……)
```

のように記述します．

ただし，int型以外はmain( )以前に定義しなければならないので，なるべくint型にしたほうが扱いやすくなります．このとき仮りに戻り値の「受け皿」がchar型でも，関数を実行するときに型変換が行なわれるため，使用時に不都合が生じないのが普通です．またint型は型名が省略できるので，プログラムの記述も軽減されます．

1-5 数値演算と演算子

Cは，BASICのように文字列の接続を数式（足し算）で表わす機能をもっていません．しかし，数値については**代入演算子**（＝）を使って，

　　a＝b＋c

のように表わせます．上式の“＋”のような一般の演算子を**算術演算子**といい，この分類に属するものは表1.2のとおりです．

数式に関してCは，むしろBASICより進歩した記述形態をもっていて，たとえば，

　　X＝X＋A

は，

　　X＋＝A

のように短縮記述を行なうことができます．このような２項演算子と代入子との組み合わせを，**複合代入演算子**といいます．この形式では，最終的な左辺は式の左辺から２項演算子を削除した状態で決まり，右辺は式全体から代入子を削除した状態が採用されます．

もっと強力なのは，

　　Y＝X
　　X＝X＋１

のように，演算後変数値を１つ進める場合に，

●表１．２　算術演算子の種類

演算の種類	算術演算記述
乗　算	a * b
除　算	a / b
剰　余	a % b
加　算	a + b
減　算	a − b
負　数	− a

Y＝X＋＋

と書くだけで，上の２式をまとめて表現できる点です．このように“＋＋”で表わすものを**インクリメント演算子**といい，“－－”で表わすものを**デクリメント演算子**と呼びます．この種の演算子は，変数の前にも後にも書くことができ，前に書いた場合は式の実行前，あとに書いた場合は式の実行後に変数の増減が行なわれます．

このような機能は非常に便利なのですが，反面，

Y＝X＋A
X＝X＋１

●表１．３　式の評価（解釈）順位

優先順	演算子	種類	結合規則
高	（　）　［　］　．　－＞	式	左から右
↑	－　￣　！　＊　＆ ＋＋　－－　sizeof　キャスト	単項	右から左
	＊　／　％	剰余算	左から右
	＋　－	加減算	左から右
	≪　≫	シフト	左から右
	＜　＞　＜＝　＞＝	関係（非等値性）	左から右
	＝＝　！＝	関係（等値性）	左から右
	＆	ビット AND	左から右
	︿	ビット EOR	左から右
	｜	ビット OR	左から右
	＆＆	論理 AND	左から右
	‖	論理 OR	左から右
	？　：	条件	右から左
↓	＝　＊＝　／＝　％＝ ＋＝ －＝　≪＝　≫＝ ＆＝　｜＝　︿＝	単純代入/複合代入	右から左
低	，	逐次評価	左から右

●表１．４　関係演算子の種類

大小関係	関係演算記述
bが大きい	a＜b
aが大きい	a＞b
bが大きいか aと等しい	a＜＝b
aが大きいか bと等しい	a＞＝b
等しい	a＝＝b
等しくない	a！＝b

●表１．５　論理演算子の種類

論理関係	論理演算記述
AND	a＆＆b
OR	a｜｜b
NOT	！a

●表１．６　ビット演算子の種類

論理関係	ビット演算記述
AND	a ＆ b
OR	a ｜ b
Ex-OR	a ︿ b
NOT	￣a

のような場合，

Y＝X＋＋＋A

と書いたのでは，式の意味がわかりにくくなります．そこで，カッコを使い，

Y＝(X＋＋)＋A

のように書くことが望まれます．マニュアルなどには式の評価順(表1.3)が書かれていて，カッコなしでもCコンパイラは混乱することなく解釈するようになっています．しかし反対に，人間のほうが迷ってしまうのではよいプログラムとはいえません．したがってできるだけカッコを使って，わかりやすくするのが本筋です．

以上の記述方法はソース・プログラムの字数を節減し，C独得のスッキリしたリストを形成する一因ともなっていますが，反対に字数が増える要素もない訳ではありません．それは，比較の際に用いられる**関係演算子**で，

X＝＝A　(BASICなら“X＝A”と書く)

のように，“＝”が２つ続くケースです．この場合，“＝”が代入演算子と混同されないよう区別しているため，このような書き方をするようになったものです．

関係演算子には表1.4のものが用意されています．左辺と左辺の関係について評価をし，成立する(真)ときは１，不成立(偽)時には０を評価結果とします．

結果が真偽で得られる演算の代表的なものとしては，**論理演算子**をあげることができます(表1.5)．このタイプの特徴は，変数が０ならば０，０でないとき１として演算が行なわれることです．

論理演算のうちビット単位で行なわれるものをとくに**ビット演算子**と呼び，表1.6の種類があります．

また，Cには結果の真偽を選択の際に利用し，２つの式のうち一つを結果として得る

数値定数の表現

XCでは数値定数を記述するとき，10進のほかに８進や16進で表現できます．

10進表現は，**数字**をそのまま並べればよいのですが，小数点以下の数値を表現する場合以外は，最上位に0を書いてはいけないという規則があります．これは，**8進**表現のとき最上位には必ず**“0”**を書かなければならないという決まりがあるためで，Cコンパイラが先頭の“0”を見て10進かどうかの区別をしているからです．

16進表現の場合には，先頭に**“0x”**を付けて８進値との相違点をもたせています．BASICでいえば，“&H”に相当するのがこれです．

たとえば10進数の10は，ほかの進数表現では

012 ……８進表現
0xa ……16進表現

のようになります．

また浮動小数点型の場合は，必ず10進数で表現することになっています．この型の場合，１より小さい数については１の位の0は省略できます．もちろん，“$\times 10^{n}$”を表わす“en”形式も使えます．

以上のいずれの表現も，先頭に置かれる文字は必ず数字(例外として“.”を含む)になっているので，Cコンパイラはこれによって，まず数値定数であるという識別ができます．その内容が何進数かというのは，その次の段階の問題です．

このように，C言語の文法には，コンパイラを作りやすくするための工夫があちこちに盛り込まれています．

〈関係演算式〉? 〈結果が1のとき採用される式〉: 〈結果が0のとき採用される式〉

形式の**条件演算子**があります．

そのほかの演算子として，シフトを行なうシフト演算子，sizeof演算子，カンマ演算子があります．

シフト演算子は左シフトを“<<”，右シフトを“>>”で表わし，左側に書かれる変数を，右側にある変数などが示すビット数だけシフトするのに用いられます．

sizeof演算子は変数などが占有しているバイト数を得るのに使われ，たとえば

```
int    A
X=sizeof(A)
```

のXは4となります．この演算子は，通常バイト数がわかりきっているデータ型にも使われます．意味のわかりにくい数値を使わないで，プログラムをわかりやすくするためです．

カンマ演算子は，1つしか式の書けないところ(たとえばfor文のカッコ内の1つの項)に複数の式を書きたいときに使われ，カンマ(,)で区切って実行順に左から並べられます．

関数演算については，次章で説明します．

1-6 ポインタ

Cが「高級アセンブラ言語」と呼ばれる理由の1つに，**ポインタ**機能をもっていることがあげられます．

ポインタは変数の**アドレス**のことで，たとえば，

```
int    *a;
```

は，直接的には整数データのアドレス値をもつ変数aを定義することになります．アドレス値は68000では4バイトで表現し，int型と同じ長さだけメモリを占有します(図1.4)．同様に，

```
char    *b;
```

はchar型のデータのアドレスを定義するもので，この場合も4バイトとられます．これらの定義で用いられる“*”はポインタ変数を示す修飾子です．

一方，式の中でデータcのアドレスは“&c”で表わせます．このため，

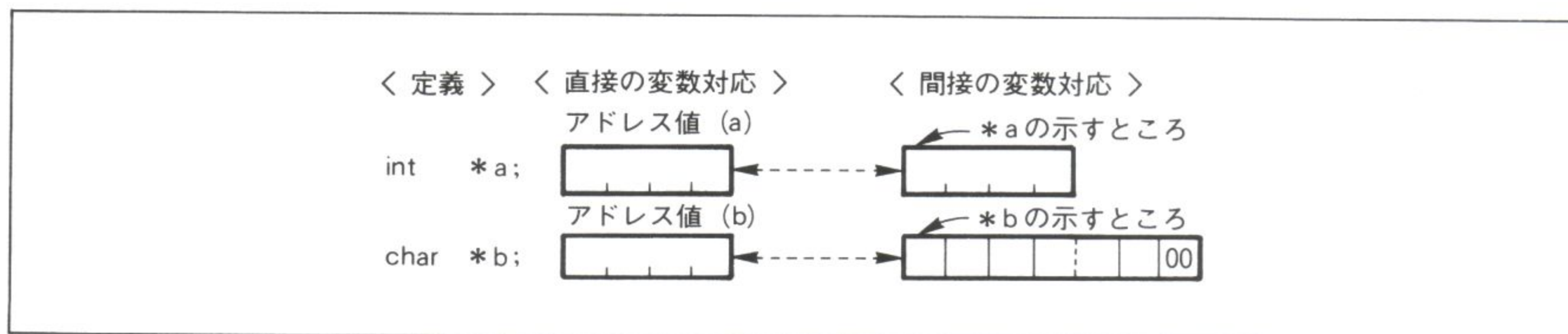
```
a=&c
```

では，aにはcのアドレスが入ります．また，

```
d=*a
```

は，aが示すアドレスに従って読み出したデータをdに入れることを表わし，この場合の“*”のことを**間接演算子**といいます．以上の結果は，

●図1．4　ポインタの基本的な考え方

〈 定義 〉　〈 直接の変数対応 〉　〈 間接の変数対応 〉
アドレス値 (a)　*aの示すところ
int　*a;
アドレス値 (b)　*bの示すところ
char　*b;
00

d＝c

と同じです．

char 型のポインタ変数(たとえば＊b)は，

b＝“abc”

のように文字列定数を右辺に書くと，その文字列のアドレスをｂに代入するという意味をもっています．この式は右辺がアドレスでないので，一見矛盾しているのですが，特別な動作を示す書き方として使われています．

1-7 データ型の変換

Ｃの関数ではパラメータとなる変数のデータ型が定義されています．したがって，理想的にはその型に合わせてデータを送り込むべきです．しかし，簡単にするため異なる型のままで関数を実行しても，たいがいの場合コンパイラが自動変換してくれるため支障は起きません．

最もわかりやすい**代入変換**は，

<変換後識別子>＝ <変換前識別子>

の形式で行なわれます．この方法は，式からもわかるように，同じデータに対して識別子が２つ存在するため定義がわずらわしいことや，変換演算のために１行だけ余分に文が必要になるなどの欠点をもっています．どちらかといえば，プログラムを読みやすくする方向には働かないので，あまり使わないほうがよいでしょう．

このような変換の意味についてよく考えてみると，それはあくまで事後の関数処理のための一時的なもので，このために変数を定義したり，代入したりするのは生産的ではありません．

そこで，こういったケースでは，次のような**キャスト変換**を利用するのがベストです．

```
sqrt ((double)x)；
```

この関数では，平方根を求めるため変数ｘは倍精度であることが必要されるのですが，

(<型名>) <変数名>

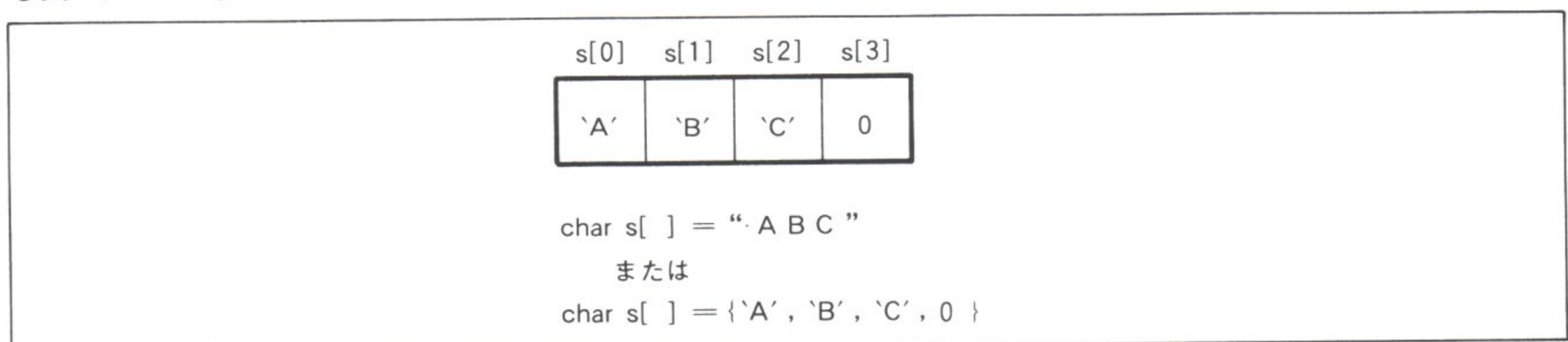
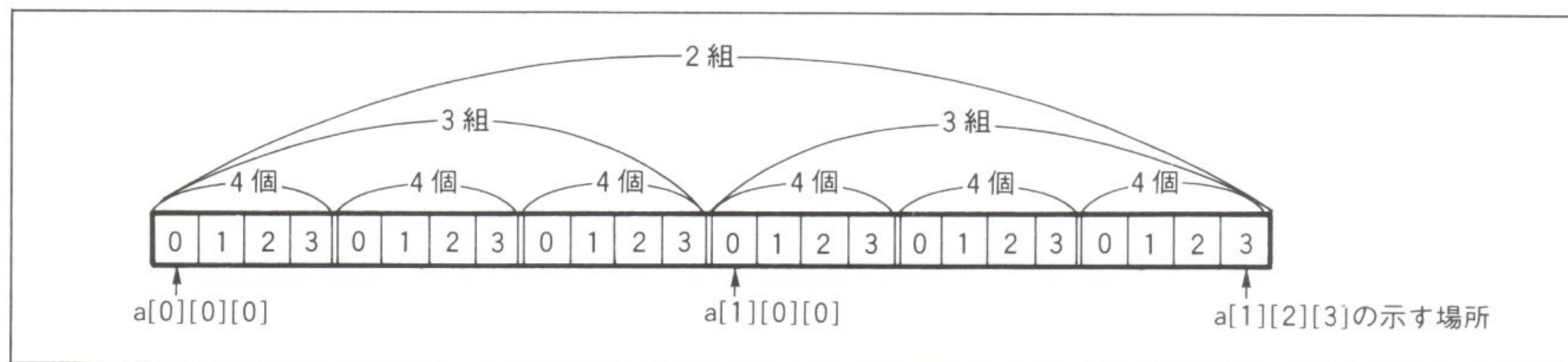
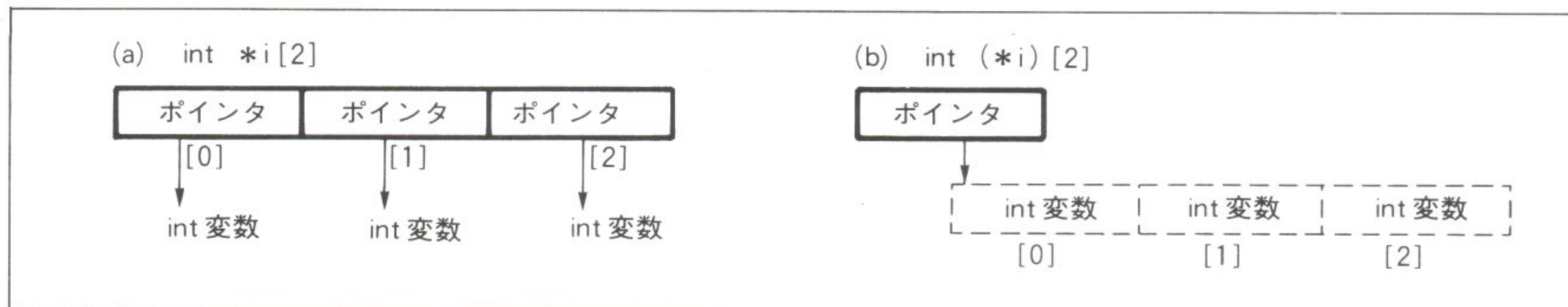
の形式で変換を目的の型に変換して与えることにより，一時的な要求を満足しています．こうすれば別途識別子を定義する必要もなく，代入変換のための文を書かなくてすみます．

1-8 配列

たとえば文字列は char 型の配列です．図1.5は，“ABC”の文字列が char 型配列に収容されたときのよ

●図１．５　文字列“ABC”の char 型配列での収容のされ方

s[0]	s[1]	s[2]	s[3]
‘A’	‘B’	‘C’	0

```
char s[ ] ＝“ A B C ”
    または
char s[ ] ＝{‘A’, ‘B’, ‘C’, 0 }
```

●図1．6　a［1］，［2］，［3］の配列の並び

●図1．7　int＊i［2］とint(＊i)［2］の違い

うすを示したもので，１字１字が１バイトずつ並べられた後にヌル(０)文字が付いて完結しています．

そのほかのデータ型についても，変換名の後に［ ］を書き**個数－１**の値を指定することにより，任意の個数の配列を定義できます．配列内の個々のデータについては０番から番号が付いており，参照するときはその番号値を［ ］内に書くことで特定のデータに限定できます．番号値は，定義するときは整数文字しか使えませんが，参照するときは変数名を書いてその変数値を利用することも可能です．

配列に初期値を与えるには，一般に，

```
char　S［ ］＝｛'A', 'B', 'C', 0｝
```

のように｛ ｝内に“，”で区切って記述できます．この例のように個数が明確なときは，［ ］内に数字を書く必要はありません．参考までに文字列については，

```
char　S［ ］＝"ABC"
```

のように直接記述する方法もあります．

ここで注意が必要なのは，関数内部で定義するauto領域の配列の場合，０以外の初期値を与えられないことです．また，auto領域の変数はスタックにとられるため，あまり大きなものを定義するとパンクしてしまいます．

配列は一次元だけでなく，［ ］を並べることで多次元のものにも対応できます．

たとえば，

```
a［1］［2］［3］
```

は，図1.6の配列に対応します．

配列の要素としては，ポインタをとることもでき，その構造によって図1.7のような書き方の違いがあります．ここで，

```
int　＊i［2］
```

は［２］が先に解釈されるため，要素数３（２に１が加算される）個の配列になります．その内訳は，int型データへのポインタ変数です．また，

```
int　(＊i)［2］
```

は( )内がコンパイラによって先に評価され，最初にポインタ変数であることが認識されます．続いて，［2］が解釈され，そのポインタは3個の要素をもつ int 型データの配列のアドレスを示すように結論づけられます．

このように，カッコを使うことによって意味が大きく変化するので，注意が必要です．

1-9 構造体の考え方

たとえばファイルなどでは，1件のデータについて複数の項目が含まれ，しかも各項目のデータ型はまちまちということがよくあります．こういったデータを扱うときは，個々の項目単位に転送などをするのは非能率で，関数にデータを引渡しするにもパラメータの数が多くなりすぎるなどの問題が生じます．

このようなときは，全項目をひとまとめにして「集団項目」として取り扱うのが便利で，それを可能にするのが**構造体**です．

図1.8(a)は構造体の構造名を宣言する例で，複数の項目をまとめて“hizuke”というタグ名を与えています．この例では各項目は同型なので“，”で区切っていますが，異なった型を混合する場合は各型ごとに“；”で仕切る必要があることは言うまでもありません．

次に，(a)の型名を使って具体的な変数(kaishibi)を定義したのが同図(b)です．ここでは初期値を設定していますが，{ } 内を省略すれば代入なしの変数にできます．初期値設定は，外部定義時のみ可能です．

このように，一度型名を宣言すれば，同じ構造のデータは何度でもそのタグ名に利用定義できます．しかし，1回限りの構造についてはタグ名を省略し，構造の宣言と構造体識別子の定義とをごちゃまぜに書

●図1．8　構造体の定義例

```
struct  hizuke          {
  ┌────┬──┐
  │    │年│  char   nen,
  │ 日 ├──┤
  │    │月│         tsuki,
  │ 付 ├──┤
  │    │日│         hi;
  └────┴──┘
                        }   ;
```

(a)　構造体による構造名の宣言例

```
struct  hizuke  kaishibi    = {
  ┌────┬──┬──┐
  │ 開 │年│90│         90,
  │    ├──┼──┤
  │ 始 │月│ 4│          4,
  │    ├──┼──┤
  │ 日 │日│15│         15;
  └────┴──┴──┘
                             }   ;
```

(b)　宣言ずみの型名を使った集団項目と初期値設定例

```
struct                  {
  ┌────┬──┐
  │ 開 │年│  char   nen,
  │    ├──┤
  │ 始 │月│         tsuki,
  │    ├──┤
  │ 日 │日│         hi;
  └────┴──┘
                        }   kaishibi;
```

(c)　構成項目の内容と集団項目の変数名を同時に宣言する例

けます(図(c))．ここで注意すべきなのは，(a)と(b)では{ }の意味が異なるということです．(b)のそれは代入する内容値を表わしています．(c)の{ }は(a)と同じで，構造体の**メンバー**(構成項目)記述を表わします．それに続く変数名は，(b)の変数名と対応しています．

構造体のメンバの参照は，構造体識別子とメンバーの識別子を組み合わせて，

<構造体識別子>．<メンバー識別子>

のデータ名で行なうことができます．たとえば(b)の月は，**kaishibi.tsuki**で表わされます．また，その構造体のアドレスがsで与えられているとき，言い換えると，

s＝&kaishibi

のとき

s－＞tsuki

は，kaishibi.tsukiと同じです．これは，

(＊s)．tsuki

とも書くことができます．sの示すアドレスとは，すなわちkaishibiのものなのでこうなるのです．

Cでは，構造体データを引数として渡すことが許されていません．したがって，構造体データを利用したいときは，代わりにそのアドレスを引数にします．関数の側では，その構造体名を使って構造体へのポインタを定義し，上記のような書き方で個々のメンバをアクセスすればよいのです．

1-10 領域の再定義ができる共用体

共用体は，宣言の仕方その他は構造体に非常によく似ていますが，すべてのメンバーがメモリの同じ位置から割り付けられる点が大きく異なります．このような性格から，共用体のサイズはメンバーの中で最も長いものと同じになります．

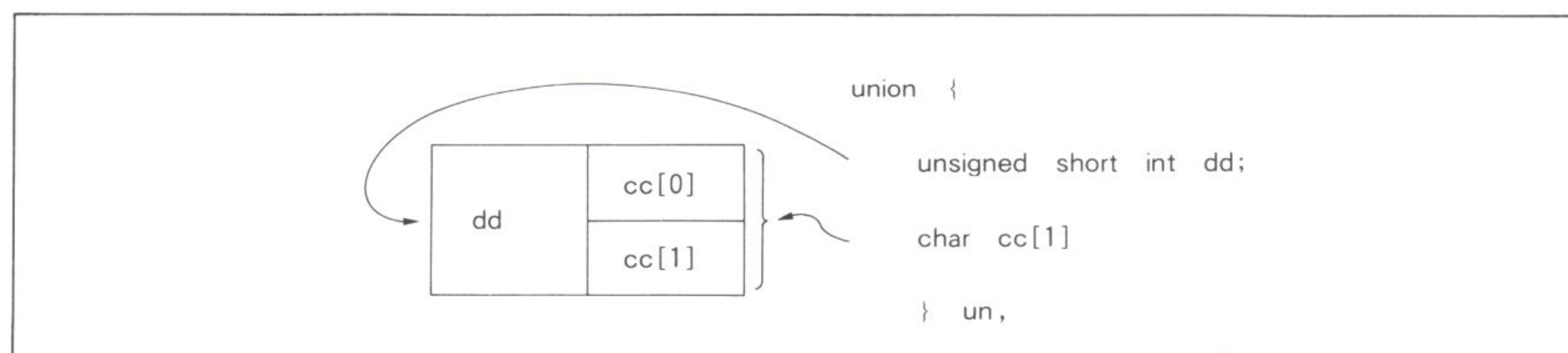
一方プログラムでは，命令行は定義ずみのデータ型によって参照するため，別な命令行で異なった定義ずみのデータ型で参照した場合，当然のことながら内容値は保証されません．共用体のために，自動的に今現在のデータ型で参照するという機能は用意されていないのです．したがってプログラマは，常に共用体に書き込んだデータ型を意識しておかなければなりません．

むしろ共用体は，同じ場所にデータを異なったデータ型で参照できることを積極的に利用し，同一ファイルに異なったフォーマット(書式)で入出力する際などに，構造体と組み合わせて利用すると便利です．その場合，各フォーマットの共通位置に，どのフォーマットかを識別する標識のようなデータを置き，プログラムでこれを判別して読み込むといったコントロールを行なえばよいのです．

また，共用体だけでも

●図1．9　共用体の定義例

```
union {
    unsigned short int dd;
    char cc[1]
} un,
```

dd	cc[0]
	cc[1]

```
union　{
        unsigned　short　int　dd;
        char　cc［1］;
        }　un;
```

のように定義して，2バイト・データ(dd)を1バイトずつ分割して，cc［0］，cc［1］として参照する(図1.9)などの利用方法があります．

なお，共用体の初期化は外部定義に限ってできます．その場合，初期値はメンバーの全変数で共用することに注意しなければなりせん．構造体のように各メンバーごとの初期値を与えるのと異なり，1つのメンバーに対して設定することになります．

上記の例でメンバーを参照するときは，たとえばddについては“un. dd”というデータ名が使えます．

構造体の参照の仕方の説明で，「構造体識別子」を「共用体識別子」と読み変えれば，まったく同じような使い方ができます．

1-11　ビット・データの扱い

ビット・データは，構造体として整数型の領域に割り付けられます．図1.10は，そのようすを示したもので，**unsigned**の次にデータ名(**メンバー識別子**)と“：”それにビット数を表わす定数式が付くのがこのタイプの特徴です．

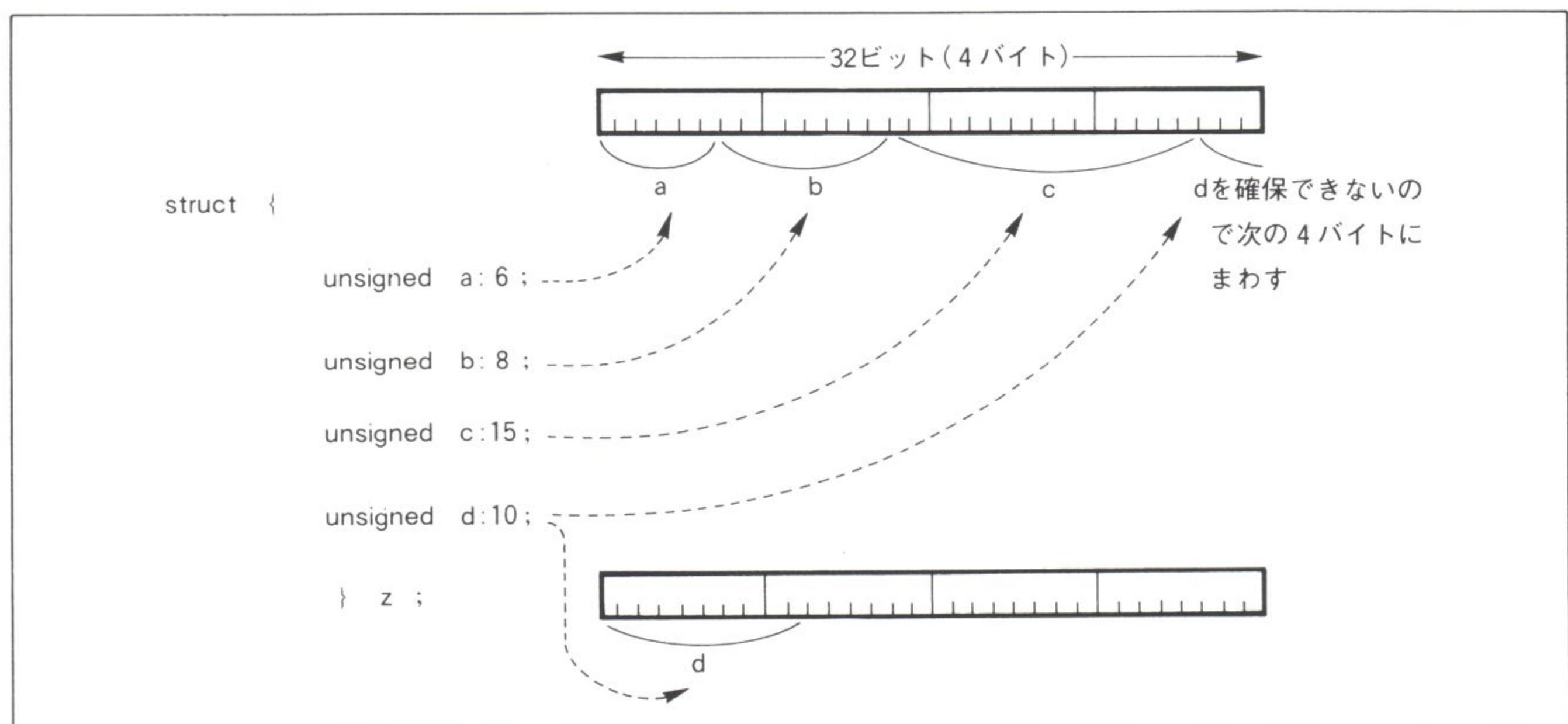
この書き方では，int型データ(4バイト)の中を上位のビットから順に割り付けしていくので，残されたビット数を新しく宣言されたビット数に配分する段階でビットが不足することがあります．このようなとき，次の4バイトにまたがって定義すると，ハードウェアの都合で処理が著しく遅くなるので，残りのビットを捨てて，新しい4バイトの先頭から割り付けを行ないます．すなわち，68000のレジスタ長は4バイトあるので，この範囲で処理を行なうのは容易ですが，それを越えると難しくなるのです．

このような境界調整は，強制的に行なわせることもできます．後続のビット・データを新しい4バイトに割り付けしたいときは，その直前に，

```
unsigned ： 0 ；
```

を定義すると，Cコンパイラは現在処理中の4バイト単位の割り付けをそこで打ち切ります．

●図1．10　構造体を使ったビット・データの割り付け

```
struct　{
        unsigned　a：6；
        unsigned　b：8；
        unsigned　c：15；
        unsigned　d：10；
        }　z　；
```

1-12 列挙型宣言

❖関連コマンド　enum

プログラムの中で，コード表のように参照される部分では，隣接して同じ性格のデータを並べることになります．こういった場面では，個々のデータは enum 型を使って定義するのが適当です．

enum 型は int 型変数の集まりで，個々の値はとくに指定がなければ 0，1，2，……と 1 つずつ増やして並べられます．

この型の定義は，

enum <タグ名> {<メンバー識別子リスト>} [<enum 識別子>]；

によって行ない，構造体と同様に上位識別子を省略した型宣言もできます．

たとえば曜日コードを，

```
enum day {
        mon= 1 ,
        tue,
        wed,
        thu,
        fri,
        sat,
        sun,
        } week;
```

のように定義すると，week.sun の値は 7 となります．

一度定義されたタグ名の引用は，

enum <タグ名> <新しい enum 識別子>；

の形態で行なうことができます．enum とタグ名までが型名に該当し，これによって定義ずみのメンバー識別子の内容が利用可能になります．

1-13 繰り返し制御文

❖関連コマンド　do 〜 while, while, for, continue, break

BASICのFOR〜NEXTやWHILEのような繰り返し制御文は，Cにも用意されています．一見ループの呼び名も似ているのですが，動作は図1.11に示すとおり微妙に異なります．

```
do {<実行文>}   while(<条件式>);
```

ループ(同図(a))では，文を実行した後，条件式が成立すると繰り返しが行なわれます．したがってBASICのWHILEの場合のように，条件によっては一度も実行されないということはありません．

もしBASICと同じような動作を希望するなら，

```
while(<条件式>) {<実行文>};
```

ループ(同図(b))を使うとよいでしょう．この書き方では，条件式は実行前に検査されます．

do〜whileもwhileも，条件式に影響するループ式をもたないので，実行文の中で直接・間接的にループ制御をしないとエンドレス・ループになります．

一方，

```
for([<初期化式>]; [<条件式>]; [<ループ式>] {<実行文>};
```

ループ(同図(c))では，文の実行前に条件式の成否が検査されるので，一度も実行されないケースが存在します．この点もBASICと反対なのですが，初期化式とループ式を省略するとBASICのWHILEループに相当する働きとなります．[]内の省略時の扱いは，初期化式とループ式に相当する部分は何もせず，条件式は常に成立とみなします．したがって条件式をカットして，エンドレス・ループを形成することもできます．

言い換えると，forループは多目的ループであって，これ1つだけマスターしておくと，do〜whileなどを知らなくてもプログラミングできます．しかし，ループの性格によっては，do〜whileなどのほうが適していることもあるので，forループに充分慣れたら次は，do〜while，whileもマスターすると，もっと良

●図1．11　Cのループ構造

(a)　do〜while文によるループ

文の実行 / while(条件) / 成立 / 不成立

(b)　for文によるループ

初期化式 / 条件 / 不成立 / 成立 / 文の実行 / ループ式

(c)　while文によるループ

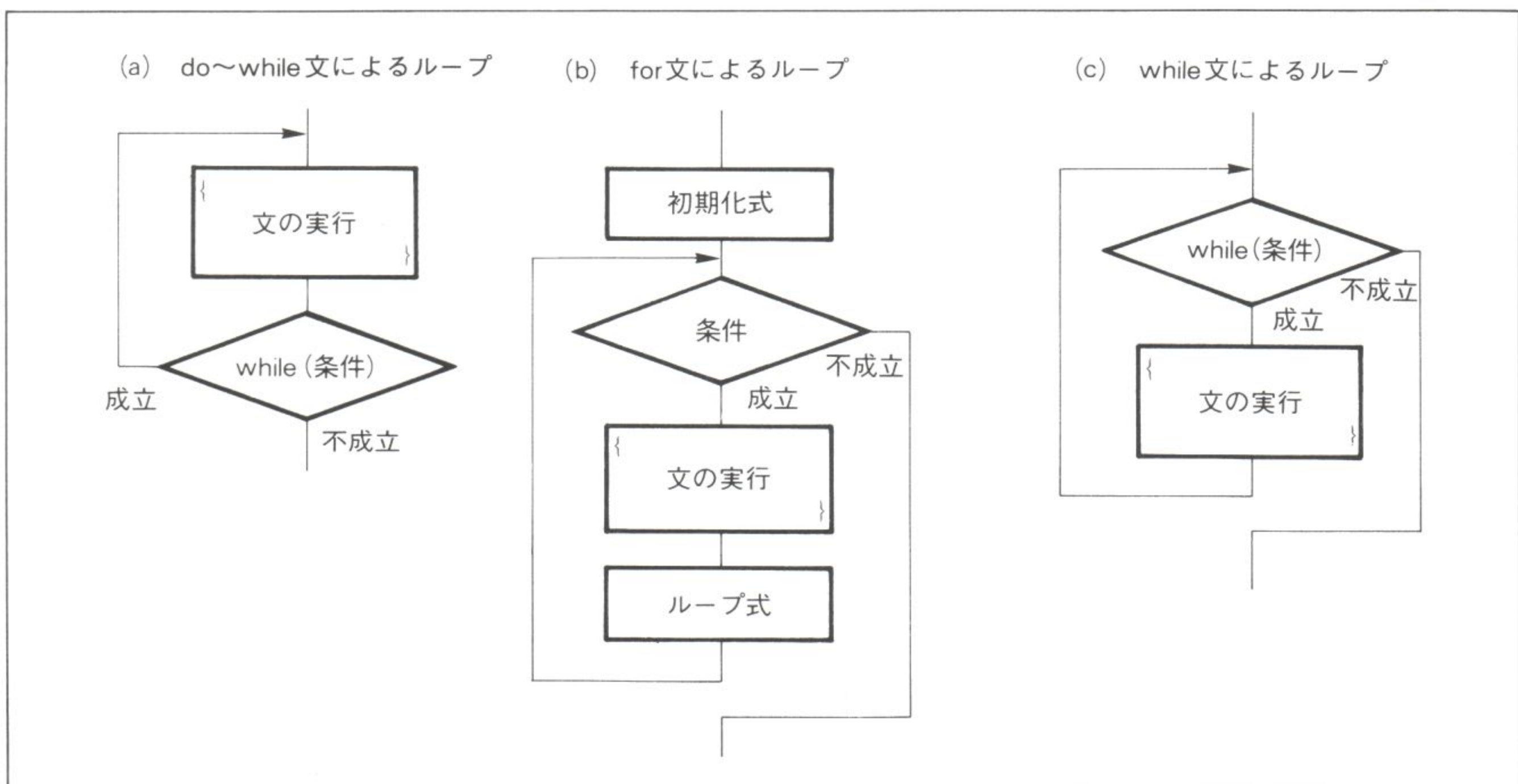
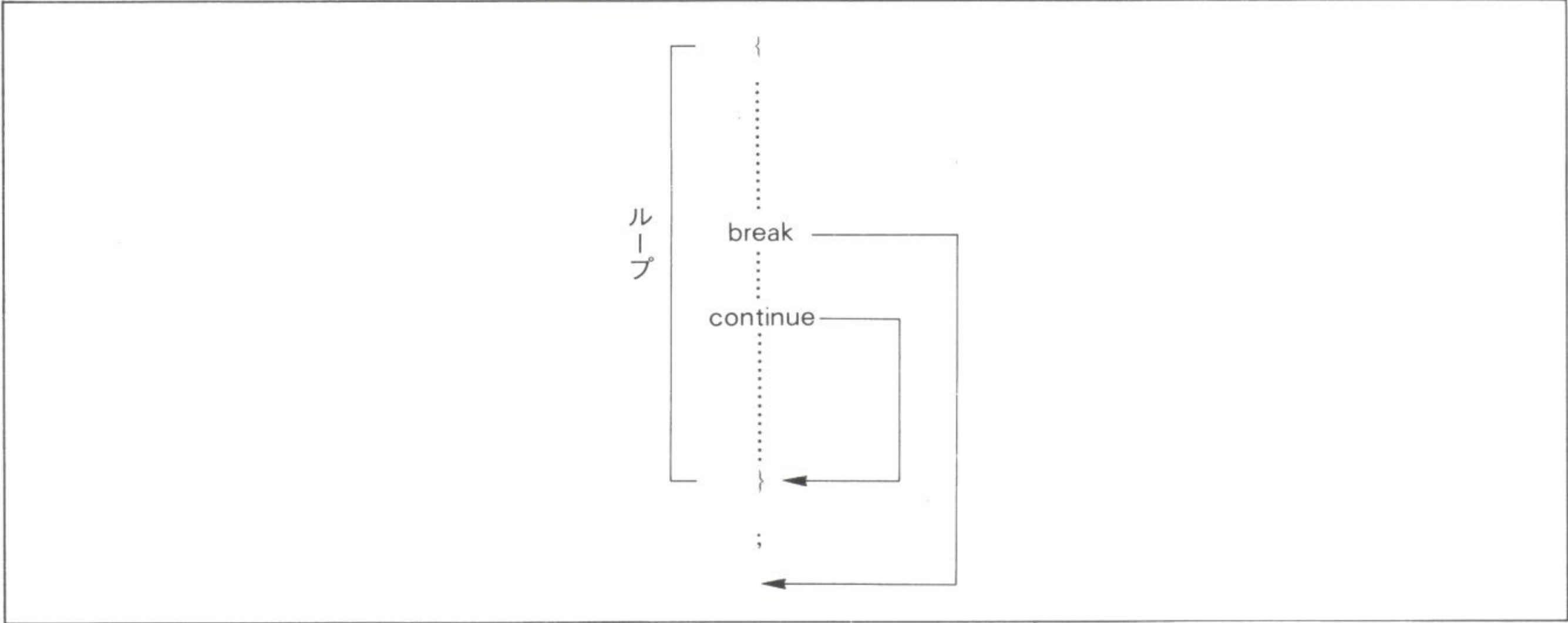
while(条件) / 不成立 / 成立 / 文の実行

●図1．12　break と continue の働き

いプログラムが書けるようになります．

図1.12の break と continue は，それぞれ実行文中に書かれる文の1つです．そのうち continue 文は以下の実行文をバイパスし，ただちに繰り返し制御に委ねるときに使われます．break 文はループの外に脱出するもので，具体的にはループ記述の次の位置にジャンプします．これらの文は，do～while，for ループのいずれにも使えます．

1-14 条件付き実行

❖関連コマンド　　if ～ else, switch

条件判断を伴なう実行のパターンには，if 文と switch 文があり，これらは BASIC のものと同様な機能をもっています．

一般によく使われるのは，

```
if(<条件式>) <成立時実行文>    else  <不成立時実行文>;
```

で，条件式は関係演算子が使われるケースがほとんどです．しかし，関係演算子だけでなく論理演算子など結果を真偽で表わすものは，ここの条件式の中の演算子として利用できます．

もう1つの

```
switch(<式>) {
    case  <定数式>:  <対応実行文>;……;
    case  <定数式>:  <対応実行文>;……;
                :
    default: <対応実行文>;……;
};
```

は，式(識別子名をそのまま書くことが多い)の評価結果(値)によって，定数式が該当する case の対応実行文から始まり“}”までの実行文が続けて実行されます．一般に1つの定数式に対する実行文は1対1で与えられることが多く，その場合個別の実行文並びの締めくくりには break が使われます．すなわち，break が実行されると，以下の実行文をバイパスして switch 文を終了させます．

switch 文の default は，その上に書かれる定数式のいずれにも該当しない場合に実行される内容を記述します．上の case 記述が働き，かつそこに break 文がないときは，当然 default 対応実行文も実行の対象となります．

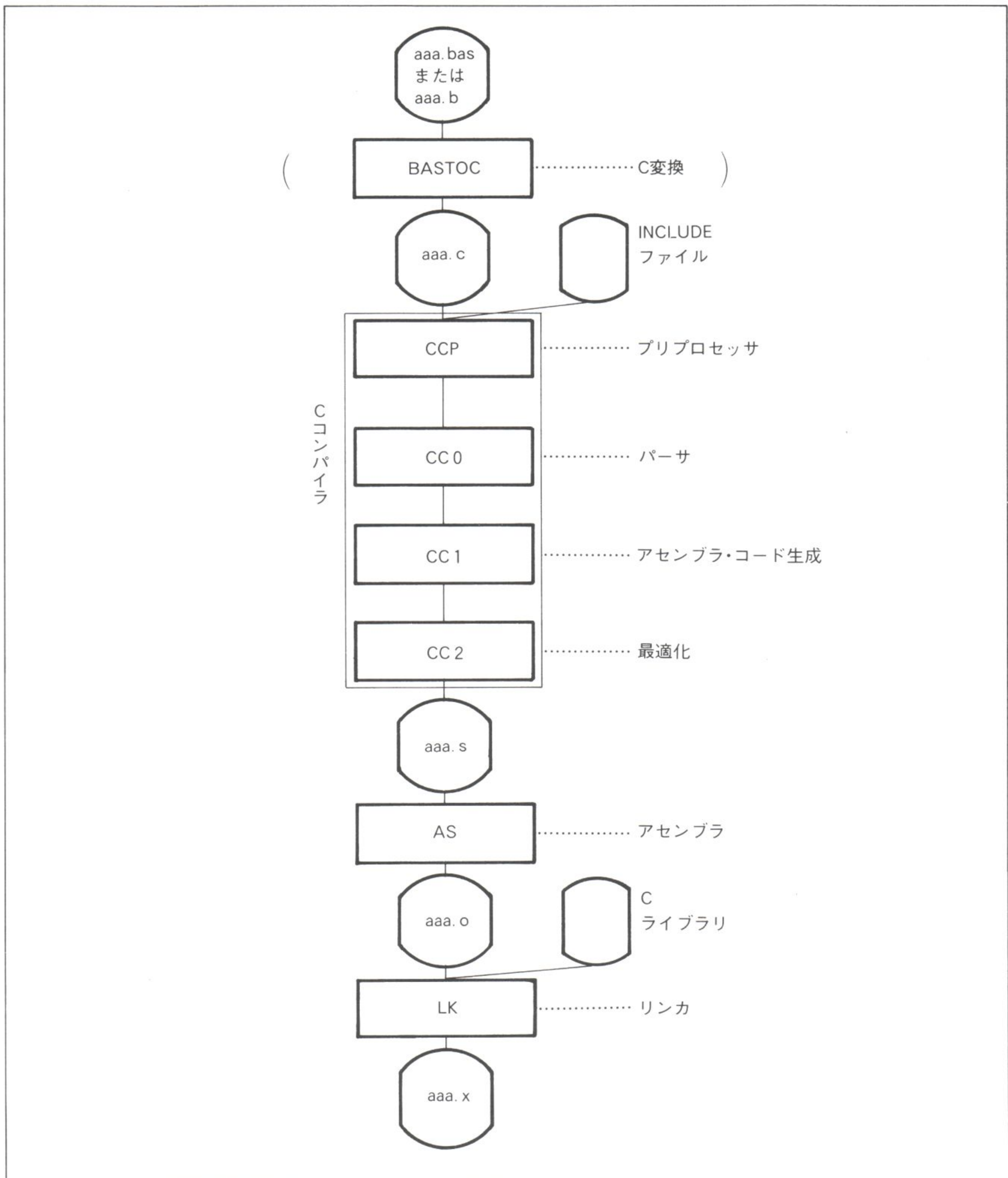
1-15 Cコンパイラ・ドライバ(cc)の使い方

Cコンパイラ・ドライバは，Cプログラムのソースから実行形式の機械語プログラム・ファイルを作成するまでのプロセスを，自動的に行なわせるようにした便利なツールです．

Cコンパイラ・ドライバが扱う範囲には，Cだけでなく，Cに変換できる文法で書かれたBASICプログラムも含まれます(図1.13)．もし入力の拡張子が“.b”または“.bas”のファイルに指定されると，最初にBC(BASTOC)が起動され，C変換が行なわれるようになっているのです．

本来の使い方は，拡張子“.c”をもつCのソース・プログラムを入力とします．この内容は，コンパイ

●図１．13　Cコンパイラ・ドライバの制御範囲（プログラム名はaaaとする）

aaa.bas または aaa.b
BASTOC ……………C変換
aaa.c
INCLUDE ファイル
Cコンパイラ
CCP ……………プリプロセッサ
CC0 ……………パーサ
CC1 ……………アセンブラ・コード生成
CC2 ……………最適化
aaa.s
AS ……………アセンブラ
aaa.o
C ライブラリ
LK ……………リンカ
aaa.x

ラによってアセンブラ・ソース形式に変換され，さらにアセンブラ，リンカを経て実行形式の機械語ファイルが作成されます．このパターンでは，BCを起動することなく，直接Cコンパイラから実行が開始されます．

Cコンパイラ・ドライバを立ち上げるには，一般的には

cc 〈プログラム・ファイル名〉

だけで充分です．単純な構造のプログラムの場合は，スイッチなどを使う必要もないことが多く，ほとんどのケースはこれで間に合います．

一連の過程で，同一ファイル名で拡張子を“.s”にした**アセンブラ・ソース・ファイル**，同じく拡張子が“.o”の**オブジェクト・ファイル**，拡張子が“.x”の**実行形式機械語ファイル**が作成されます．もし入力ファイルがBASICソースならば，拡張子が“.c”の変換後の**Cソース・プログラム・ファイル**も作成されます．これらのファイルは，元のソース・ファイルに比べて大きなスペースを占有するので，終了後に中間的なファイルはdelコマンドで消去しておくほうがディスク管理上望ましいでしょう． スイッチを利用するときは，コマンド記述では原則としてプログラム・ファイル名の前に指定します．スイッチの種

●表1．7　Cコンパイラ・ドライバのスイッチ

スイッチ	働き
/B	BASTOCのみ実行する．
/C	プリプロセッサに入力テキスト中の注釈を削除しないよう指示する．
/P	プリプロセッサ出力リストをファイル（拡張子“.p”）に出力する．
/V	プリプロセッサ出力を標準出力パスに出力する．プリプロセッサのみで終了するときに指定する
/U〈識別子〉	(識別子は20個まで可能) プリプロセッサに，指定された識別子を含む＃unde fine文を削除させる．
/D〈識別子〉	(識別子は20個まで可能) プリプロセッサに，指定された識別子に対する＃define文を追加させる．
/I〈ディレクトリ名〉	インクルード・ファイルのパスを“A：￥include”から変更したいときに指定する．
/O	CC2（オプティマイザ）によって，CC1から出力されたアセンブラ・ソース・ファイルの内容を最適化する．
/K〈ドライブ名〉	アセンブラ・ソース・ファイルを出力するドライブを指定する．
/G [〈ファイル名〉]	アセンブル・リストを指定されたファイルに出力する．/Gのみのときは，“prn”の拡張子が付けられる．
/Q〈ドライブ名〉	オブジェクト・ファイルを出力するドライブを指定する．
/S	アセンブラまで終了する．
/F	リンク時に高速化するプログラムCHSH.Xを使用せず，メモリを節約する．
/A〈ライブラリ・パス〉	A：￥LIB以外のライブラリ・パスを指定するときに使う．
/Y	DOSLIB.A，IOCS.Aライブラリを使用するときに使う．
/W	BACIC，Aライブラリを使用するときに使う．
/J [〈マップ・ファイル名〉]	リンク・マップを出力するファイル名を指定する．/Jだけのときは，標準出力パスに出力される．
/Z〈ファイル名〉	実行形式機械語ファイル名をとくに指定したいときに使う．
/L	リンカを実行しない．

類は表1.7のとおりです．

なお，Cコンパイラは**CCP**(プリプロセッサ)，**CC0**(パーザ)，**CC1**(アセンブラ変換)，**CC2**(最適化)の各段階に分かれて実行されます．そのあとアセンブラ，リンカを経て実行形式のプログラムができあがります．この過程でスクリーンには各段階ごとにメッセージが表示され，現在時点の段階が把握できます．

スイッチ	働き
/E	エラー出力を標準出力パスでなく指定ファイルに変更する．
/M	CCの処理順に従って作成されるファイルの作成年月日をチェックし，新しいものがあればその部分の処理をバイパスする．
/R	CCの実行経過の表示．
/X	CCの実行をシミュレート，表示し，処理は行なわない．
/H〈インダイレクト・ファイル〉	CCコマンドのスイッチなどをインダイレクト・ファイルで与えるとき，ファイル名を指定する．
/T〈作業用ディレクトリ〉	コンパイル実行時の作業用ディレクトリを，カレント・ディレクトリまたは環境で指定されたパス以外にしたいとき指定する．
/k	関数実行時にスタック・サイズが充分かどうかチェックさせる．
/f〈テキスト・ファイル名〉	テキスト・ファイルからアセンブラに変換するとき，対応関係がわかるように，テキストの内容をコメントして挿入する．
/g	同一文字データを同一ラベルで参照するようにする．
/h〈ヒープ・サイズ〉k	大きなデータ領域が必要なとき，KB単位に指定する．
/i〈識別子の長さ〉	Cの識別子の長さを32字より小さくしたいとき指定する．ただし8未満は指定できない．
/j〈case数〉	1つのswitch文に対するcase数を257以上に拡張したいとき，1,000未満の範囲で指定する．
/m〈メンバー数〉	構造体のメンバーの数を128より大きくしたいとき，1,024までの範囲で指定する．
/n〈登録数〉	識別子の登録数を2,000から他の数(400～65,535)にしたいとき指定する．
/s〈スタック・サイズ〉K	スタック・サイズは通常64KBとられているが，変更したいとき4KBからシステムで可能な値まで指定できる．
/t〈ストリング長〉	ストリングの長さを256から拡張したいとき32,767までの値で指定する．
/w〈レベル〉	構文チェックのレベルを0から拡張したいとき3までの範囲で指定する．

第2章

INCLUDEファイルと メーカー提供関数の概要

2-1 INCLUDEファイルの参照

Cによるプログラミングは，メーカー提供の関数を利用しないで行なうことは考えにくい状況にあります．

関数はどこかで定義されていなければならないので，メーカーではそのための定義と，関連グローバル変数，標準サポート構造体一式をまとめて**INCLUDEファイル**として提供しています．これら定義の内容はかなりの量にのぼるため，実際は機能別に分けてファイルに収容されています．XCでは，INCLUDEディレクトリの下にこれら各ファイルが配置されています．

INCLUDEファイルは，ユーザーが作ることもできますが，その場合はユーザーの使用するディレクトリ内に配置するのが普通です．

INCLUDEファイルの参照は，Cプログラム中で# include文の出現した位置で行なわれ，指定されたファイルの内容がプリプロセッサによりそこに挿入されます．このとき，基本的にはCコンパイラ起動時に指定されたディレクトリ，環境変数のincludeで指定されたディレクトリの順で検索されますが，ファイル名がクォーテーション(")で囲まれているときは，カレント・ディレクトリの内容が優先されます．このような記述は，ユーザ定義のINCLUDEファイルを引用するのに適しています．メーカー提供のファイルは，“<”と“>”で囲んで指定します(図2.1)．

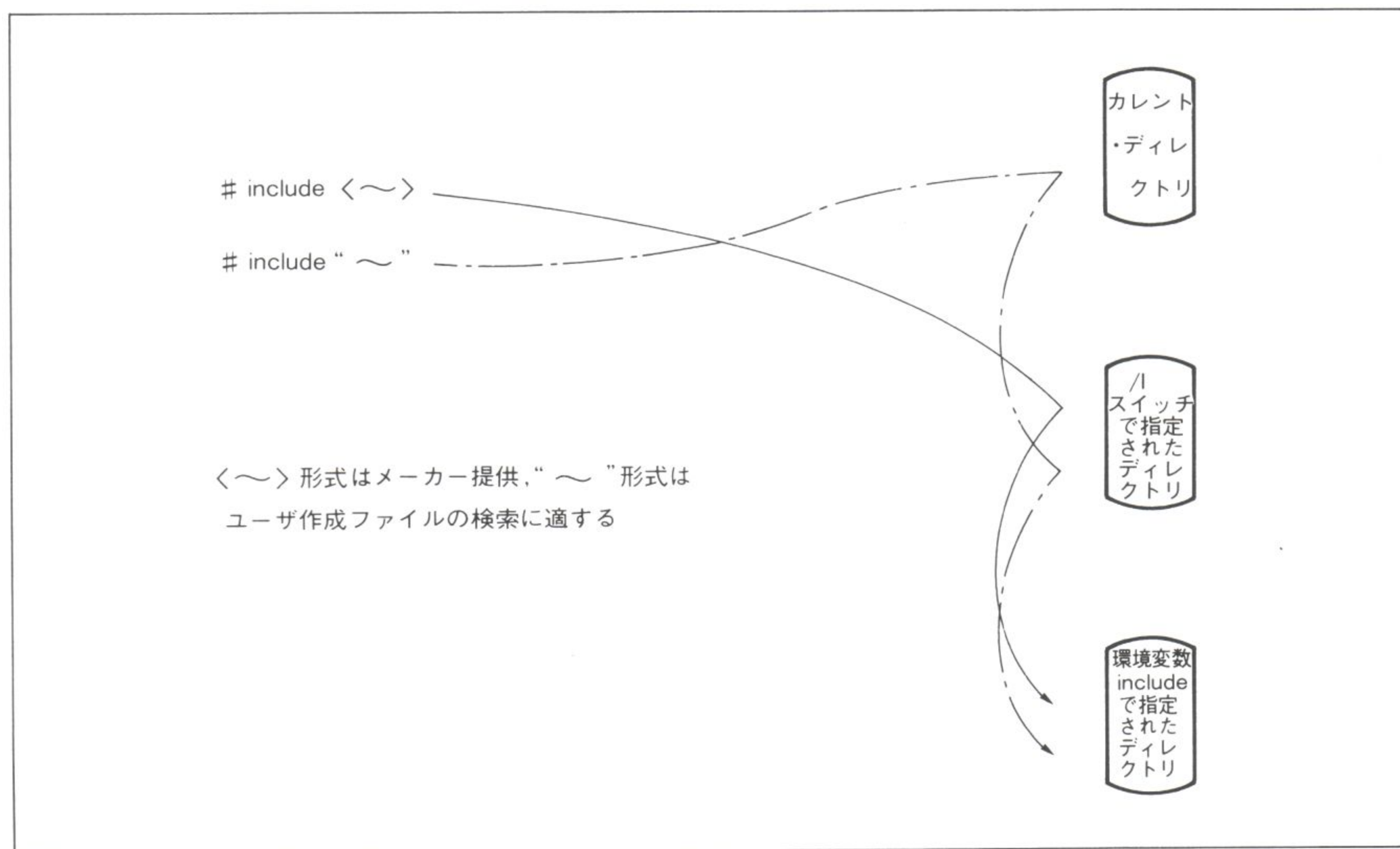
XCの一般用INCLUDEファイル名のリストを表2.1に，BASICからCに変換する際に使われるもののリストを表2.2にそれぞれ示します．

X68000のメーカー提供関数は非常に数が多く，参照のための定義記述を収容している INCLUDE ファイルも多種に及びます．

本章ではこれらの参照の仕方とメーカー提供関数のアウトラインを紹介します．また，ユーザー独自の INCLUDE ファイルの作り方についても，実例をあげて説明します．

メーカー提供関数の個々の説明はマニュアル中に膨大な量で書かれているので，本章では詳細には触れません．むしろ，全体の関連とか特徴ある関数などについてのガイドに徹することにしたいと思います．

関数の具体的な応用例については本章でも一部取り上げていますが，次章で入門的なものから応用に至るまでを紹介しています．

●図2．1　＃include 文のファイル名の指定方法によるディレクトリの検索順

●表2．1　XCでサポートされている一般用INCLUDEファイル

ファイル名	内容
assert. h	実行時チェック・マクロの定義
class. h	データ型式のシンボルの定義
conio. h	コンソール入出力を行なう関数の宣言
ctype. h	文字チェックおよび文字変換のマクロ定義
direct. h	ディレクトリ操作の関数宣言
doslib. h	Human68Kのファンクション・コールの関数宣言
error. h	エラー・コードのシンボル定義
fcntl. h	低水準入出力関数用定数の宣言
fctype. h	ctype. hで定義されているマクロの関数版の宣言
fefunc. h	浮動小数点演算のシンボル定義
float. h	(算術ライブラリの）浮動小数点演算用定数の宣言
io. h	ファイル操作と低水準入出力関数の宣言
iocslib. h	X68000のIOCSコールの関数宣言
jtctype. h	日本語ライブラリ用関数の宣言
jstring. h	
limits. h	各データ形式のリミット・チェック
math. h	算術ライブラリ用関数の宣言
process. h	プロセス制御関数の宣言
setjmp. h	setjmp関数用構造体などの宣言
signal. h	signal関数用定数などの宣言
stat. h	statおよびfstat関数用定数などの宣言
stdio. h	標準入出力用関数などの宣言
stdlib. h	標準ライブラリ関数の宣言
string. h	文字列関数用文字列操作関数の宣言
time. h	time関数用構造体などの宣言
timeb. h	ftime関数用構造体などの宣言
utime. h	utime関数用構造体などの宣言

●表2．2　BASIC→C変換に使われるINCLUDEファイル

ファイル名	内容
audio. h	ADPCMの制御関連の関数の宣言
basic. h	X-BASIC本体組み込みの関数の宣言
basic0. h	画面処理関係他の関数の宣言
graph. h	グラフィック制御関連の関数の宣言
image. h	イメージ制御関連の関数の宣言
mouse. h	マウスの制御関連の関数の宣言
music. h	FM音源の制御関連の関数の宣言
sprite. h	スプライトの制御関連の関数の宣言
stick. h	ジョイスティック入力用の関数の宣言

＊シャープ「PRO-68KCライブラリマニュアル」

2-2 データ型の変換関数

数値同士ならば，ある型のデータを別なデータに変換することは簡単です．つまり，代入形式の記述においてデータは自動的に変換されてしまうからです．

その半面，数値から文字列，あるいはその逆を行なうときは，関数を利用しなければなりません．このときのデータ型と使用関数の関係は，表2.3のとおりです．

この表を見るとわかるように，データ型によってはサポートされていない部分があったり，同じデータ型について複数の関数が用意されているものがあります．

複数用意されている関数同士は少しずつ動作が異なっていて，たとえば**strtol**は８進とか16進文字列も扱えるのに対し，**atol**は10進文字列だけを想定しています．また，**ecvt**，**fcvt**，**gcvt**の違いは，前二者が指数のない形式のみなのに対し，gcvtは指数なし形式で扱えない場合は指数形式にします．ecvtとfcvtの違いは，前者が全体の桁数を指定するのに対し，後者は少数点以下の有効桁数を指定する点です．

その他の文字列に変換する関数は，基数を指定することによって好みの進数値に変換できます．基数には２～36が使えます．

●表２．３　文字列との間の変換関数の一覧表

型	文字列から	文字列へ
char		
int	atoi	itoa
short int	atow	
long int	atol，strtol	ltoa
unsigned char		
unsigned int		uitoa
unsigned short int		uwtoa
unsigned long int		ultoa
float		
double	atof	ecvt, fcvt, gcvt

（atof：stdlib.h，その他：math.h）

2-3 数値演算関数

数値演算のうち演算子を用いるものについては前章で説明しましたが，その中にないパターンについては，表2.4の関数などを用いて対応することになります．同表のものは，ほとんどの関数で引数として double型が定義されているのが特徴です．このように最大限の装備をしておけば，演算の精度によって関数を使い分ける必要がないのです．

演算にはエラーがつきものですが，これら数値演算ライブラリ中の関数では，エラーが発生すると**matherr**関数（ユーザ定義）が呼び出されます．このとき，引数として図2.2の構造体**exception**のポインタが与えら

●図２．２　構造体 exception

```
Struct  exception  {
        int     type
        char    *name ;
        double arg1, arg2 ;
        double retval ;
} ;
```

●表2．4　数値演算関数

●math.h

関数名	数値演算の内容
acos	アークコサインを求める
asin	アークサインを求める
atan	アークタンジェントを求める
atan2	アークタンジェントを求める
cabs	複素数の絶対値を求める
ceil	最小の整数値を求める
cos	コサインを求める
cosh	ハイパボリックコサインを求める
except	例外処理を行なう
exp	指数関数を求める
fabs	浮動小数点値の絶対値を求める
floor	最大の整数値を求める
fmod	剰余を求める
frexp	浮動小数点値を仮数部と指数部に分ける
hypot	直角三角形の斜辺の長さを求める
ldexp	浮動小数点値の指数部を求める
log	自然対数を求める
log10	10を底とした対数を求める
matherr	エラー処理を行なう
modf	浮動小数点値を整数部と小数部に分ける
pow	べき乗を求める
sin	サインを求める
sinh	ハイパボリックサインを求める
sqrt	平方根を求める
tan	タンジェントを求める
tanh	ハイパボリックタンジェントを求める

れます．

ここでのエラー処理は，一般に構造体のメンバーを参照して行なわれます．たとえばエラーの種類を参照するには，構造体アドレスを＊ｓで受けたとき，

```
s－＞type
```

のように書きます．

2-4　文字，文字列を操作する関数

Ｃでは，文字列は文字の配列としてとらえています．したがって文字を操作する関数は，文字列を操作するのに使えないことはありません．しかし，文字列の操作は１字１字に対して繰り返す必要があることや，末尾(ヌル：０)の判定を必要とするなど，一般にプログラムが複雑になることは致し方ありません．このためＣでは，よくある文字列処理のパターンを関数化しているので，それで対応できるものならば利用するのがよいでしょう．

文字を扱う関数は，大別して文字の種類を検査するもの(表2.5)と文字の種類の変換を行なうもの(表2.6)の２種類に分類できます．前者は，文字のコードを調べて，目的の類型であるかどうかによって戻り値をセットします．このとき目的の型でなければ，戻り値には0がセットされます．後者は，検査してその結果が変換対象ならば目的の文字種に変換するもので，引数で指定した文字が直接操作されます．

文字列を扱うものには，いろいろな種類があります．単純なものではコピー(表2.7)，比較(表2.8)，少し高度なものでは文字の順序を入れ換えるもの(表2.9)などがあります．また文字列の連結が必要なときは，文字列追加関数(表2.10)を使えばBASICでの文字列の加算と同様なことができます．

文字種の変換を文字列全体に対して行ないたいときは，そのための関数(表2.11)も用意されているので，

●表2．5　文字の種類を検査する関数の一覧表

●fctype.h（マクロ版は ctype.h）

関数名	戻り値=検査文字となる文字種
isalnum	英数字
isalpha	英文字
isascii	アスキー文字
iscsym	識別子（C言語）
iscsymf	識別子の先頭文字（C言語）
iscntrl	制御文字
isdigit	10進数
isgraph	表示可能文字（空白を除く）
islower	英小文字
isprint	表示可能文字（空白を含む）
ispunct	句読点
isspace	空白文字（009～0x0D）
isupper	英大文字
isxdigit	16進数

●表2．7　文字列をコピーする関数の一覧表

●string.h

関数名	コピー動作
strcpy	普通のコピー
strdup	alloc で割り付けた領域にコピーする
strncpy	コピー先の余った字数をヌル(0)で埋める

●表2．8　文字列同士を比較する関数の一覧表

●string.h

関数名	比較動作
strcmp	大文字と小文字の区別あり
strcmpi	大文字と小文字の区別なし
strncmp	大文字と小文字の区別，字数指定あり

●表2．6　文字種の変換を行なう関数の一覧表

●fctype.h（マクロは fctype.h）

関数名	変換対象文字種	変換後の文字種	備考
toascii	非アスキー文字	アスキー文字	
tolower	英大文字	英小文字	
toupper	英小文字	英大文字	
_tolower	英大文字	英小文字	変換対象文字種以外の動作は保証されない
_toupper	英小文字	英大文字	

●表2．9　文字列内の文字の順序を入れ換える関数

●string.h

関数名	入れ換え動作
strrev	逆順にする

●表2．10　文字列を追加する関数の一覧表

●string.h

関数名	追加位置
strcat	後方
strncat	後方（字数指定あり）
strins	前方

●表2．11　文字列全体で変換を行なう関数の一覧表

●string.h

関数名	変換対象文字種	変換後の文字種
strlwr	英大文字	英小文字
strupr	英小文字	英大文字

●表2．12　文字列の内容を入れ換える関数の一覧表

●string.h

関数名	置き換えの範囲
strnset	先頭から指定された字数
strset	末尾のヌル(0)を除く全文字

●表2．13　ファイル名を扱う関数の一覧表

●string.h

関数名	ファイル名に対する処理
strmfe	ファイル名の拡張子を置き換える
strmfn	ファイル名を作成する（4変数から）
strmfp	ファイル名を作成する（2変数から）
stcgfe	ファイル名から拡張子を取り出す
stcgfn	ファイル名からノードを取り出す

これらを利用すれば1文字用の文字種変換関数を使う必要がありません．文字列の内容を別な文字で入れ換えする表2.12のような関数も使えます．

ファイルを処理するために，プログラム内でファイル名を合成する必要があるときのためには，表2.13の関数が便利です．

こういった既存の関数では対応しきれないときは，プログラマが必要な処理を行なわせるようプログラミングしなければなりませんが，その際には対象文字列についての位置情報が必要になります．表2.14の関数は，このようなときに使用するものです．

●表2．14　ファイル名を扱う関数一覧表

●string.h

関数名	文字列に対する検査内容
strchr	指定文字の最初の位置を調べる
strcspn	指定文字列の最初の位置を調べる
strlen	文字列の長さを調べる
strpbrk	指定文字列を探す
strrchr	指定文字の最後の位置を調べる
strspn	指定文字列以外の最初の位置を調べる
strtok	トークンを探す
strbpl	文字列からポインタ配列を作成する

●表2．15　日本語処理関数の一覧表

関数名	日本語処理の種類
	jfctype.h
iskana	カナ文字またはカナ句読点の判定を行なう
iskpun	カナ句読点の判定を行なう
iskmoji	カナ文字の判定を行なう
isalkana	英文字またはカナ文字の判定を行なう
ispnkana	英句読点またはカナ句読点の判定を行なう
isalnmkana	英数字またはカナ文字の判定を行なう
isprkana	空白を含む表示可能文字の判定を行なう
isgrkana	空白を除く表示可能文字の判定を行なう
iskanji	漢字の1バイト目の判定を行なう
ikanji2	漢字の2バイト目の判定を行なう
	jstring.h
jstrncmp	漢字を含む2つの文字列を比較する
jstrchr	漢字を含む指定文字の最初の位置を調べる
jstrrchr	漢字を含む指定文字の最後の位置を調べる
jstrcmp	漢字を含む2つの文字列を比較する
	jfctype.h
jiszen	全角文字の判定を行なう
jisl0	全角空白文字または JIS水一水準記号の判定を行なう
jisl1	JIS第一水準漢字の判定を行なう
jisl2	JIS第二水準漢字の判定を行なう
jisalpha	全角英文字の判定を行なう
jisupper	全角英大文字の判定を行なう
jislower	全角英小文字の判定を行なう
jisdigit	全角数字の判定を行なう
jiskata	全角カタカナ文字の判定を行なう
jishira	全角ひらかな文字の判定を行なう
jiskigou	全角句読点の判定を行なう
jisspace	全角空白文字の判定を行なう
hantozen	半角文字を全角文字に変換する
zentohan	全角文字を半角文字に変換する

日本語では漢字が使われますが，漢字は2バイト・コードのため特殊な処理が必要です．表2.15は，日本語文字あるいは文字列を操作する関数の一覧です．

●制御文字を特定の文字に変換する関数

iscntrl関数の応用に，制御文字を特定の文字に変換する関数を作る例を図2.3に示します．ここではその関数をmain( )によってテストしていますが，注意したいのは，文字コード0～0xffの範囲でパラメータを与える際に，for文で指定する繰り返し条件がchar型変数では与えられない点です．なぜなら，char型では0xffの次は0になってしまうので，変数名を“cd”とすると，

```
cd <= 0xff;
```

は歯止めにならないからです．仕方がないので，文字コードはint型で生成したもの(d)を使用しています．

このプログラムのテスト結果は図2.4のとおりです．

●図2．3　制御文字を“.”に置き換える関数とテスト用メイン部

```
#include <stdio.h>
#include <ctype.h>

main()                                                        ┐
{                                                             │
     int d, cnt=0;                                            │
     char s[64];                                              │
     for (d=0; d<0x100; d++ ) {                               │
          s[cnt] = iscn (d);       ―変換結果を文字列に並べる  │ 関数をテストする
          if (cnt++ == 63) {       ｛64字に達したら文字列を   │ メイン部
               s[64] = 0;          ｛出力する                 │
               puts (s);                                      │
               cnt = 0;                                       │
          }                                                   │
     }                                                        │
}                                                             ┘

iscn (x)                                                 ┐
unsigned char x;                                         │
{                                                        │
     int r;                                              │ 制御文字を“.”に置き換える
     r = iscntrl (x);     ―制御文字かどうか検査          │ 関数
     if (r == 0)                                         │
          return x;       ｝制御文字でなければそのまま返す │
     else                                                │
          return '.';     ｝制御文字なら“.”を返す         │
}                                                        ┘
```

●図2．4　制御文字を“.”に置き換える関数のテスト結果
(CZ-8PD3プリンタ(非漢字)で印字したので，漢字コード部分はスペースにしている)

```
................................ !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[¥]^_`abcdefghijklmnopqrstuvwxyz{|}¯.
                                 。「」、・ヲァィゥェォャュョッーアイウエオカキクケコサシスセソ
タチツテトナニヌネノハヒフヘホマミムメモヤユヨラリルレロワン゛゜
```

2-5 標準入出力用関数のすすめ

「関数言語」と呼ばれるCには，まさに「つくだ煮」ができるほどたくさんの関数が用意されています．その中でもとくに入出力に関するものはバラエティに富んでおり，一つ一つ覚えるのは大変です．

標準入力，あるいは標準出力という概念は，とくにデバイスの種類を意識しないでプログラムを記述するもので，指定がなければキーボードが入力，スクリーンが出力デバイスとして用いられます．また，リダイレクト記号(<：入力用，>：出力用)を使って任意のデバイスに接続変更ができるので，たとえば都合によってディスクから入力したり，プリンタで出力したりすることが実行段階で自由に行なえます．

このため，他の入出力関数を知らなくても，標準入出力用関数さえ使えれば普通のプログラムに要求される処理には充分対応ができます．

具合のよいことには，標準入出力にはオープン，クローズという手続きがいらないので，プログラムはその分だけ簡単になります．反面データは，スクリーンに表示できる内容(文字列)であることを前提にしているため，整数データなどを含むことができません．もし数値データを利用したければ，文字列との間の変換関数の助けが必要になります．

表2.16は標準入出力関数を一覧表にまとめたものです．表によってわかるとおり，入出力は1字単位あるいは行文字列単位のいずれでも可能です．このほかに，書式(フォーマット)に従った項目単位の入出力もできます．

●表2．16　標準入出力関数の一覧表

●stdio. h

関数名	入出力動作
fgetchar( )	標準入力から1文字読み込む
getchar( )	標準入力から1文字読み込む（マクロ定義使用可）
gets	標準入力から1行読み込む
scanf	標準入力からフォーマット・データを読み込む
fputchar	標準出力へ1文字書き込む
putchar	標準出力へ1文字書き込む（マクロ定義使用可）
puts	標準出力へ1行書き込む
printf	標準出力へフォーマット・データを書き込む

データ行はスクロール画面に順序よく表示できる性格のものでなければならず，いわゆる「順編成ファイル」として取り扱われます．このため，任意の行にバックしたりはできません．

2-6 ファイル・ハンドルについて

一般に，プログラムは目的に応じて，複数の入力・出力ファイルをもっています．言い換えると，個々の入出力ファイルを識別して処理する仕組があるので，このために**ファイル・ハンドル**が使われます．

1つのファイル・ハンドルはオープンするときに与えられ，クローズするときに開放されます．使用中のハンドルは他のファイルで使わないようにするので，衝突することがないのです．

ハンドルの中には最初から割り付けられているもの(表2.17)があり，これらについてはオープン手続きなしで入出力が行なえます．

標準入出力関数でオープンする必要がないのは，これらのファイル・ハンドルを利用しているためです．その他の入出力関数，とりわけ低水準入力関数では常にファイル・ハンドルを意識して処理を行なうことになります．次節で述べるストリーム入出力関数では，直接ファイル・ハンドルを扱うことは少ないのですが，故意に同じハンドルを引き継ぐなどのケースで利用することがあります．

●表2．17　定義ずみのファイル・ハンドル

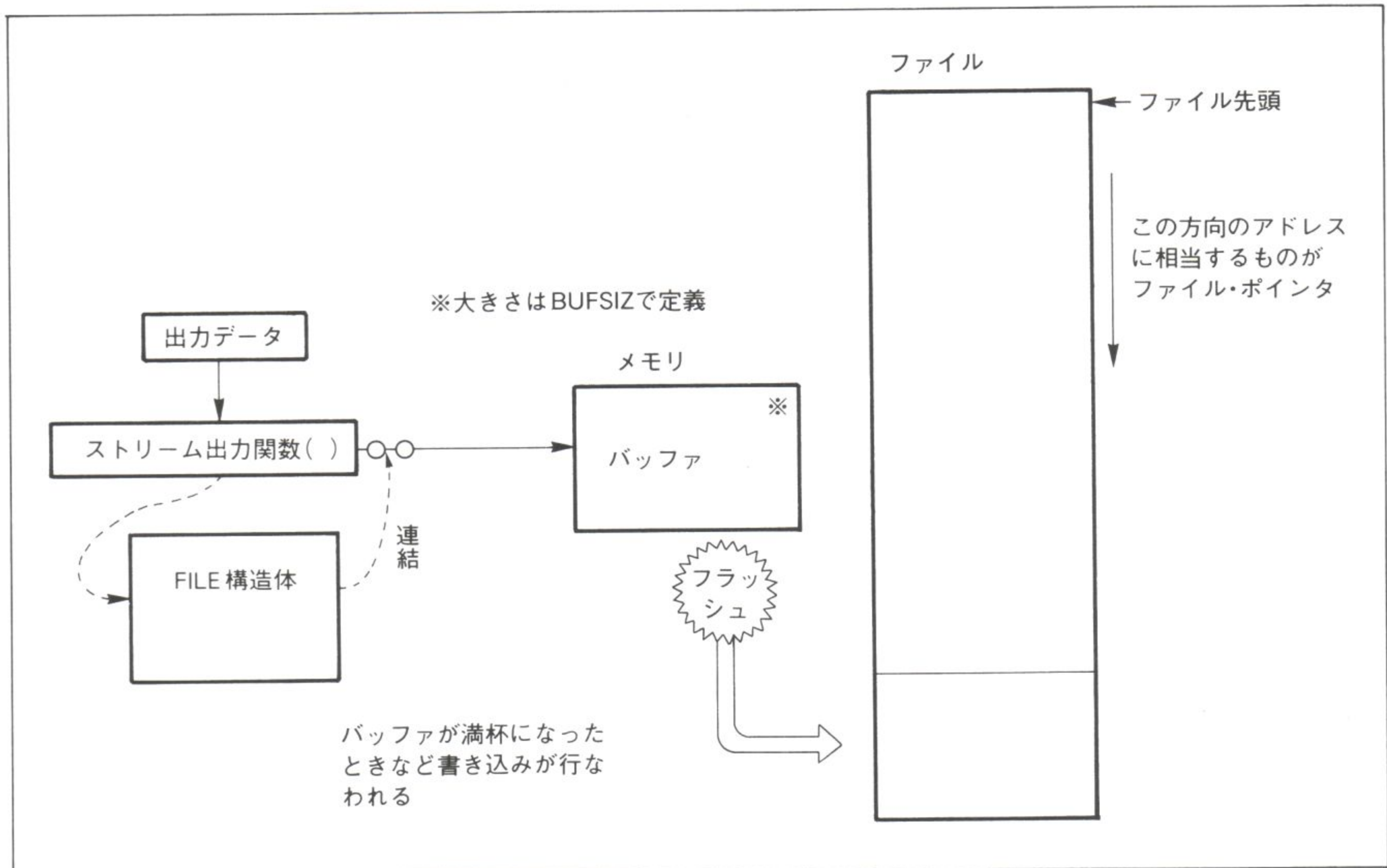
ストリーム	ハンドル	対応する入出力デバイス
stdin	0	標準入力（通常はキーボード）
stdout	1	標準出力（通常はスクリーン）
stderr	2	標準エラー出力（通常はスクリーン）
stdaux	3	標準補助入出力
stdprn	4	標準プリンタ出力

2-7 ディスク用入出力関数

ディスクなどのファイルは，「**ストリーム**」という概念で処理できます．図2.5はストリームの物理的な対応関係を示したもので，表2.18の**入出力関数**は，暗黙にこのような制御構造を利用しています．しかし，この表の関数を利用する際に，図の概念を知らなくてもほとんど困ることはありません．標準入力，標準出力もストリームの一形態にすぎないからです．

ストリームの処理はファイルをオープンすることから始まり，このときFILE構造体との連結，ファイル先頭(追加の場合は追加位置の先頭)への位置づけが行なわれます．

あとは，読み取りまたは書き込みが行なわれるたびに**ファイル・ポインタ**が次のデータ位置まで移動し，

●図2．5　ストリームによる出力の概念

●表2．18　一般的なストリーム入出力関数の一覧表

● stdio.h

関数名	入 出 力 動 作
fopen	ストリームをオープンする
fileno	ストリームのファイル・ハンドルを求める
fdopen	ファイル・ハンドルでストリームをオープンする
getc	ストリームから1文字読み込む(マクロ定義使用可)
fgetc	ストリームから1文字読み込む
fgets	ストリームから文字列を読み込む
getw	ストリームから short 型データを読み込む
getl	ストリームから int 型データを読み込む
fread	ストリームから固定長データを読み込む
fscanf	ストリームからフォーマット・データを読み込む
feof	指定ストリームが EOF かどうか調べる
fputc	ストリームへ1文字書き込む
putc	ストリームへ1文字書き込む(マクロ定義使用可)
fputs	ストリームへ文字列を書き込む
putw	ストリームへ short 型データを書き込む
putl	ストリームへ int 型データを書き込む
fwrite	ストリームへ固定長データを書き込む
fprintf	ストリームへフォーマット・データを書き込む
ferror	ストリームの入出力時のエラーを調べる
clearerr	ストリームのエラーインジケータをクリアする
fclose	ストリームをクローズする
fcloseall	すべてのストリームをクローズする

連続したファイル処理ができるようになっています．

このとき，実際のデータは**バッファ**を経由してプログラムのデータ領域とファイルとの間を移動(コピー)します．たとえば出力の場合，データはバッファに書かれ，バッファが満杯になった時点で自動的にディスクに転送されます．このような実際の書き込みのことを，**フラッシュ**といいます．フラッシュは，出力ファイルをクローズするときも，バッファにデータが残っていると自動的に行なわれます．いわばプログ

●表2．19　ストリームの特殊操作関数の一覧表

●stdio.h

関数名	入出力動作
fflush	ストリームをフラッシュする
flushall	すべてのストリームをフラッシュする
fseek	ストリーム・ポインタを移動する
freopen	ストリーム・ポインタを再割り付けする
ungetc	読み込んだ文字を入力ストリームに戻す
setbuf	ストリーム・バッファの制御を行なう
setnbuf	ストリーム・バッファを変更する
setvbuf	ストリーム・バッファを変更する
ftell	ファイル・ポインタの位置を求める
rewind	ファイル・ポインタをファイルの先頭へ移動する
fmode	ファイルの変換モードを変更する
sprintf	文字列へフォーマット・データを書き込む
sscanf	文字列からフォーマット・データを読み込む

ラムでは，バッファという「窓枠」を通してファイルを見ているからで，バッファそのものは直接扱いません．

ストリームに対しては，表2.19のような特殊操作を行なう関数も用意されています．ファイルがシーケンシャルにアクセスされるだけなら最後までバッファやファイル・ポインタを意識する必要はないのですが，同表の関数の多くはこれらを意識的に操作して，思いのままの位置で入出力を行なえるようにするためのものです．

2-8　低水準入出力関数，コンソール入出力関数

標準入出力あるいはストリームによる入出力は，システムがバッファ操作を代行してくれるため，ユーザは背後で何が行なわれているかをほとんど意識する必要がありません．

これに対し，**低水準入出力関数**(表2.20)では，基本的にシステムで面倒みてくれるのはファイルとバッファ間の転送までです．したがってストリーム・データの書かれたファイルを読んでも，そのままではブロック単位の「かたまり」が得られるだけです．

このことは低水準入出力ではストリーム・ポインタが用意されておらず，**ファイル・ポインタ**しかないことからも明らかで，読み書きするデータは直接ディスクの物理アドレスと対応しています．すなわち，直接アドレス(ファイル・ポインタ)を指定してアクセスするような用途に向いているといえます．

低水準入出力関数がディスクの**ランダム・アクセス**に向いているのに対し，**コンソール入出力関数**(表2.21)は，水準の低さは同様なのですが，キーボード入力とスクリーン出力(表示)に適している点が異なります．

コンソール系のデバイスでは，一般にCR／LFで行の終わり，改行を表わします．長いテキストでは複数行にまたがることもありますが，このような特殊性がcgets, cputsなどの入出力関数に反映されていま

●表2．20　低水準入出力関数の一覧表

●io.h

関数名	低水準入出力動作
close	ファイルをクローズする
creat	ファイルを新規に作成する
dup	ファイル・ハンドルをコピーする
dup 2	ファイル・ハンドルを強制的に再割り付けする
eof	ファイルがEOFかどうか調べる
lseek	ファイル・ポインタを移動する
open	ファイルをオープンする
read	ファイルからデータを読み込む
tell	ファイル・ポインタの位置を求める
write	ファイルへデータを書き込む

●表２．21　コンソール入出力関数の一覧表

●conio.h

関数名	コンソール入出力動作
cgets	コンソールから文字列を読み込む
cprintf	コンソールへフォーマット・データを書き出す
cputs	コンソールへ文字列を書き出す
cscanf	コンソールからフォーマット・データを読み込む
getch	コンソールから１文字読み込む
eetche	コンソールから１文字読み込みエコーバックする
kbhit	キー入力の有無を調べる
putch	コンソールへ１文字書き出す
ungetch	読み込んだ文字をコンソールに戻す

す．すなわち，cgetsはCR／LFを検出するか指定文字数に達すると入力を完了させ，LR／LFをヌル文字(＄０)に置き換えます．また，出力命令のcputsではCR／LFのコントロールを含む文字列を前提にしているため，文字列の末端の＄０を処理しても自動改行はせず，単にストッパとして認識するだけです．

その意味では，標準入出力関数のputsを使うほうがプログラムとしては簡単になるのですが，制御文字の本来の意味を活かした表示制御ができないというデメリットがあります．Ｃの入出力関数は，やたらに種類が多くて初心者はどれを使えばよいのか迷ってしまいますが，このような用途の違いを知っておけば悩むことはありません．

なお，コンソール入出力関数では，実行時のリレダイレクトができます．

2-9　ディレクトリとディスク・ファイル操作関数

ここでは，ディスク・ファイルの環境や状態などを取り扱う関数について述べます．

その中で**ディレクトリ**関係のものでは，表2.22の関数が用意されています．ここで，getcwd(戻り値としてカレント・ディレクトリの値を参照する)以外は，Human68Kのコマンドと同じ名前でかつ同じ機能を

●表２．22　ディレクトリ操作関数の一覧表

●direct.h

関数名	ディレクトリ操作の内容
chdir	ワーキング・ディレクトリを変更する
getcwd	ワーキング・ディレクトリを求める
mkdir	子ディレクトリを作成する
rmdir	子ディレクトリを削除する

●表２．23　ファイル操作関数の一覧表

●io.h

関数名	ファイル操作の内容
access	ファイルの許可設定を調べる
chmod	ファイルの許可設定を変更する
chsize	ファイルの大きさを変更する
filelength	ファイルの長さを調べる
fstat	ファイル・ハンドルのステータス情報を求める（stat.h）
isatty	文字デバイスかどうか調べる　文
mktemp	仮のファイル名を作成する
rename	ファイル名を変更する
setmode	ファイルの変換モードを設定する
stat	ファイル名からファイルのステータス情報を求める（stat.h）
unlink	ファイルを削除する

もつものが用意されています．すなわち，プログラム中でカレント・ディレクトリを変更したり，子ディレクトリを新設したり，削除したりできます．

表2.23の**ファイル操作関数**の中にも，rename のように Human68K コマンドと共通のものがあり，システム・プログラム中のサブルーチンがCでも利用できるようになっています．

同表の関数で「ファイルの許可設定」とは，書き込み可，読み出し可の設定です．このほかに，ファイル・サイズの参照または変更，ファイル・ステータス( )更新日・サイズなどを含む)の参照などを行なう関数が用意されており，一時ファイル用のファイル名生成(mktemp)，ファイル削除関数(unlink)などもあります．

これらの機能を使えば，たとえばプログラム中で作業用のディレクトリを作り，その下に一時ファイルをこしらえて，用がすんだら消去するという一連の処理を自動化できます．

2-10　メモリ，バッファを操作する関数

1台のパソコンで同時に走るプロセスが1つしかなければ，メモリは目いっぱい使えます．しかし，せっかく複数のプロセスが走れるようになっているのですから，状況に応じて縮小拡大ができるようにしておいたほうが，同時実行のチャンスを逃がさないことにつながります．これはあくまでデータ領域についていえることで，たとえばエディタならばテキストを展開する領域などのように，縮めてもそれなりの運用ができる場合に限定されます．

このようなメモリの領域は，int などの宣言で確保した部分(クローバル変数またはローカル変数)とは異なり，**ヒープ**と呼ばれています．

●表2．24　メモリ割り当て関数の一覧表

●stdlib.h

関数名	メモリ割り当て操作
allmem	使用可能なすべてのメモリを確保する
bldmem	メモリ・ブロックをKB単位で確保する
calloc	配列の領域を割り当てる
chkml	未使用メモリ・ブロックの最大のサイズ(バイト)を求める
free	割り当てずみのメモリ・ブロックを解散する
getmem	指定サイズのメモリを確保する
getml	指定サイズのメモリを確保する
malloc	メモリ・ブロックを割り当てる
realloc	割り当てずみのメモリ・ブロックのサイズを変更する
rblk	ブレーク値を初期状態に戻す
rlsmem	確保したメモリを解放する
rlsml	確保したメモリを解放する
rstmem	使用しているメモリ・ブロックをすべて解放する
sblk	ブレーク値をリセットしてヒープ領域を拡張する
sizmem	使用可能なメモリ・ブロックのサイズ(ワード)を求める

●表2．25　バッファ操作関数の一覧表

●string.h

関数名	バッファ操作
memccpy	バッファ内文字列を他のバッファにコピーする
memchr	バッファ内の文字列を検索する
memcmp	2つのバッファ内の文字列を比較する
memcpy	バッファ内文字列を他のバッファにコピーする
memset	バッファ内を指定文字列で初期化する
movedata	バッファ内文字列を他のバッファにコピーする
repmem	指定メモリ・ブロックを複数回コピーする
setmem	バッファ内を指定文字列で初期化する
swmem	2つのメモリ・ブロックを入れ換える
movmem	バッファ内文字列を他のバッファにコピーする

表2.24は，メモリ割り当てに関係する関数の一覧です．

メモリ領域を取得する(たとえばmallocなどを使う)には，事前にchkmtなどで空きを調べておくと取得に失敗する心配がありません．また，不要になった領域は，free関数などでただちに開放することがメモリの有効利用につながります．表中のsbrkは使用にあたって注意が必要な関数で，拡大だけでなく負値を指定することで領域の縮小もできますが，その後他のメモリ割り当て関数を使用した場合の動作が保証されていません，したがって，この関数は敬遠したほうがよく，同じことはreallocにさせるのが適当です．reallocではブロックのアドレスは変化しますが，内容が保存されます．

表2.25に示すバッファ操作関数は，メモリ・ブロック内のデータをコピーしたり，比較や検索などを行なうための関数です．

とくにバッファ内容のコピーはいろいろなバリエーションをもっていて，単に指定バイト数分だけ行なうもの(memcpy)，指定されたストッパ文字でも停止できるもの(memccpy)のほかにmovedata, movememがあります．movedataとmovememはmemcpyに似ていますが，機能は劣っているので注意が必要です．

このように，類似の関数が無秩序に存在するところが，Cの欠点の1つといえます．

2-11 サーチ，ソート関数

表2.26に**サーチ，ソート関数**の一覧を示します．

昇順に並んだ配列から目的のデータを探すとき，配列の前半と後半に分けて先頭のデータを調べ，存在する可能性のある側をさらに半分ずつに分けて絞り込む方法を採用すると，かなり大量のデータの中からでも高速に検索できます．このように，次々に半分に絞り込んでいくサーチの方法を，**バイナリ・サーチ**といいます．XCではバイナリ・サーチのためにbsearch関数が用意されています．bsearchでは，比較のためのユーザ関数が呼び出されます．

サーチに限らず，データは処理に都合のよい順序に並んでいることがプログラムの簡単化や高速化につながります．このための並べ換えの処理を，**ソート**といいます．

ソートは配列の要素の前後関係を比較して行なわれますが，要素の内容が単純なときは該当データ型のもの(たとえばlqsrt)を使うのが簡単です．また，要素の内容が複数の項目からなっていて，そのうちの一部だけがキーになるようなケースでは，qsrtを使って比較部分をユーザ関数で行ないます．このときユーザ関数で比較結果を反対にすれば，降順に並べることもできます．このほかqsrtでは，複数キーや複雑な大小判定によるソートにも対応が可能です．

文字列を要素とする配列のソートを行なうstrsrtは，インクルード・ファイルとしてstring. hを使います．この関数は**バブル・ソート**という簡単なアルゴリズムを使っているので低速です．大量データを扱うときにはqsrtで，ユーザ関数による比較を行なう方法を採用することが望まれます．

●表2．26 サーチ，ソート関数の一覧表

●stdlib.h

関数名	サーチ，ソートの種類
bsearch	バイナリ・サーチ
qsort	クィック・ソート
fqsort	float型のデータをソート
dqsort	double型のデータをソート
lqsort	long型のデータをソート
sqsort	short型のデータをソート
tqsort	ポインタ型のデータをソート
strsrt	文字列データをソート（string.h）

2-12 プロセス制御関数

プロセスとは，Human68K下における実行単位のことで，今現在実行しているプログラムもプロセスの1つです．また，パイプ処理時の個々のプログラムも，それぞれプロセスであることはいうまでもありません．

表2.27は**プロセス制御関数**を一覧にしたものです．

プロセスは普通，コマンドの実行により立ち上げられ，固有のプロセスIDが与えられます．その値は，getpid関数で参照することができます．

1つのプロセスから，子プロセスを起動したり，終了させたりすることによって，ユーザ・プログラムからプロセスの流れをコントロールできます．

プロセスには必ず終わりがあって，一般に実行の経過により，プロセス側で正常終了(_exit関数を使う)，異常終了(abort関数を使う)のいずれかで停止させます．プロセス側に問題がなくても，**アボート・シグナル**の受信により中止もできます．その場合，前もってsignal関数で設定しておけば，シグナルを無視する(アボートさせない)ことも可能です．

●表2．27　プロセス制御関数の一覧表

●process.h

関数名	プロセス制御の内容
abort	プロセスを異常終了する（stdlib.h）
execl	引数リストによって子プロセスを実行する
execle	引数リストと環境によって子プロセスを実行する
execlp	引数リストとPATH変数によって子プロセスを実行する
execlpe	引数リスト，PATH変数，環境によって子プロセスを実行する
execv	引数配列によって子プロセスを実行する
execve	引数配列と環境によって子プロセスを実行する
execvp	引数配列とPATH変数によって子プロセスを実行する
execvpe	引数配列，PATH変数，環境によって子プロセスを実行する
exit	プロセスを終了する（stdlib.h）
_exit	バッファをフラッシュせずに子プロセスを終了する(stdlib.h)
getpid	プロセスIDを求める
signal	シグナル割り込み操作を行なう（signal.h）
spawnl	引数リストによって子プロセスを実行する
spawnle	引数リストと環境によって子プロセスを実行する
spawnlp	引数リストとPATH変数によって子プロセスを実行する
spawnv	引数配列によって子プロセスを実行する
spawnve	引数配列と環境によって子プロセスを実行する
spawnvp	引数配列とPATH変数によって子プロセスを実行する
system	Human68Kのコマンドを実行する（stdlib.h）

●表2．28　関数によるプロセス起動時の動作の違い

関数名	PATH変数の使用	引数の渡し方	子プロセスの実行環境
execl spawnl	×	引数リスト	親プロセスの環境を引き継ぐ
execle spawnle	×	引数リスト	最後の引数として子プロセスの環境テーブルが渡される
execlp spawnlp	○	引数リスト	親プロセスの環境を引き継ぐ
execv spawnv	×	引数配列	親プロセスの環境を引き継ぐ
execve spawnve	×	引数配列	最後の引数として子プロセスの環境テーブルが渡される
execvp spawnvp	○	引数配列	親プロセスの環境を引き継ぐ
execlpe	○	引数リスト	最後の引数として子プロセスの環境テーブルが渡される
execvpe	○	引数配列	最後の引数として子プロセスの環境テーブルが渡される

プロセス制御関数の多くは子プロセスの起動にかかわるもので，これらの具体的な違いについては，表2.28に示すとおりです．

2-13 時間情報操作関数

時間情報を扱う関数を表2.29に示します．

時間の値はグリニッジ標準時を中心に，整数値またはtimeb構造体で取得できます(timeまたはftime)．グリニッジ標準時から地方時(東京時)への変換は，localtime関数で行ない，この場合は整数(timeで取得

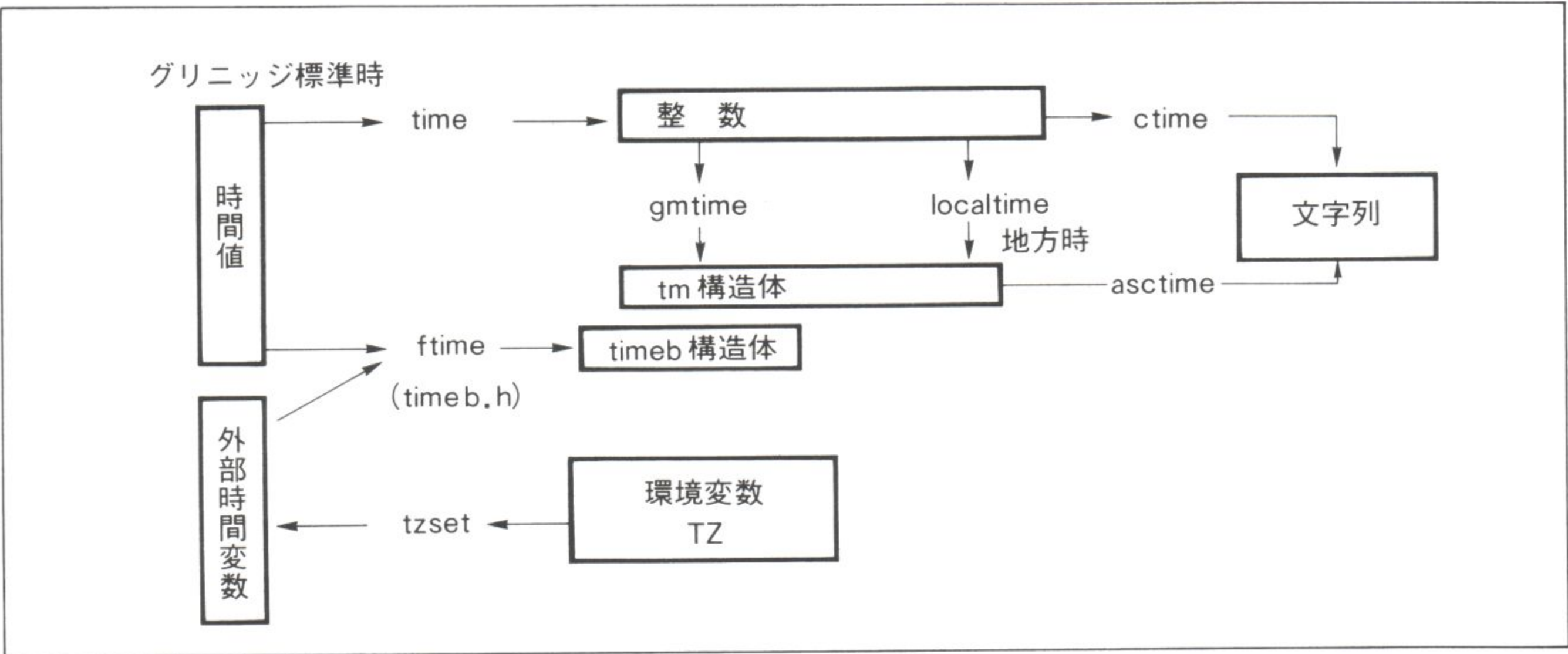
●表2．29　時間情報を処理する関数の一覧表

●time.h

関数名	時間情報の処理
asctime	時間情報を文字列に変換する
ctime	時間をint型整数値から文字列に変換する
ftime	システム時間を調べる（timeb.h）
gmtime	時間を整数値から構造体に変換する
localtime	地方時への変換
time	システム時間を調べる
tzset	環境時間変数から外部時間変数を設定する
utime	ファイル更新日付を設定する（utime.h）

●表2．30　tm構造体のフォーマット

tmのフィールド	格納される内容
tm_sec	秒
tm_min	分
tm_hour	時間（0～24）
tm_mday	日付（1～31）
tm_mon	月（0～11，0が1月）
tm_year	年（現在の年から1900を引いた数）
tm_wday	曜日（0～6，0が日曜日）
tm_yday	1年を通した日付（0～365，0は1月1日）
tm_isdst	夏期時間調整が有効の場合は0，それ以外は0

●図2．6　時間情報を処理する関数の相互関係

した値)から tm 構造体(表2.30)への変換になります．整数値，tm 構造体のデータ値は，それぞれ ctime，asctime 関数によって文字列に変換され，表示などに使えます．

これらの処理に際しては，一部で外部時間関数(グローバル変数として定義)が参照されます．この値は tzset 関数によって環境変数 TZ からセットされます．

以上の各関数の関係を整理したのが図2.6です．

これらのほかに，ファイルの更新日付をセットする utime 関数が用意されています．

2-14 その他の関数

表2.31に，これまで説明していなかった関数の一覧を掲げます．これらの関数は種々雑多ですが，環境変数の値を求める getenv，セットする putenv のように，2つの関数のコンビネーションが目立ちます．エラー表示をする perror なども，使ってみると便利な関数の1つです．この関数は標準エラー出力に対してメッセージを出力するもので，ユーザ定義メッセージに続いて，エラー番号対応のシステム・メッセージが表示されます．

BASIC 関連の関数は，X-BASIC で使用しているサブルーチンを XC でも使えるようにしたものです．また，IOCS コール，DOS コールはアセンブラで使われるサービス・ルーチンで，XC からも利用できます．これらの関数はたくさんあって．そのリストを掲載するだけでかなりのページ数が必要です．したがって，優先度が低いこともあり本書ではカットしました．

●表2．31　その他の関数一覧表

関数名	機能	
abs	int 型整数値の絶対値を求める	(stdlib.h)
wabs	short 型整数値の絶対値を求める	(stdlib.h)
assert	論理エラーをテストする	(assert.h)
getenv	環境変数の値を求める	(stdlib.h)
longimp	セーブされたスタック環境をリストアする	(setjmp.h)
perror	エラーメッセージを表示する	(stdio.h)
putenv	環境変数の値を修正または追加する	(stdio.h)
rand	擬似乱数を発生させる	(stdio.h)
setjmp	スタック環境をセーブする	(setjmp.h)
srand	擬似乱数を初期化する	(stdlib.h)
swad	データの上位/下位バイトを入れ換える	(stdlib.h)
swaw	データの上位/下位ワードを入れ換える	(stdlib.h)
BASIC 関係	(省略：basic.h など)	
IOCS コール	(省略：iocslib.h)	
DOS コール	(省略：doslib.h)	

2-15 マクロ定義，条件付きコンパイル

“#”が先頭に付く命令記述は，C のプリプロセッサがコンパイルに先立って処理するためのものです．必須の# include についてはすでに説明したとおりですが，このほかにも使って便利なものが用意されています．

●マクロ定義

define はマクロを定義するときに利用されます．C のマクロでは，アセンブラの場合のように数行またがって定義するということはあまり行ないませんが，長い記述内容を短い記述で置き換えるとか，そのままでは意味のわかりにくい数字について仮のデータ名で意味をもたせるようなときに利用すると便利です．

たとえば，前もって

```
#define   UC unsigned char
```

と定義しておくと，以下“unsigned char”は“UC”と略記できます．また，

```
#define ONNA 2
```

の定義により，JIS で制定されている性別コード２(女)を利用するとき，“ONNA”と記述できるためプログラムがわかりやすくなります．

マクロのもう１つの形態は，

#define 〈識別子〉(〈引数リスト〉)［〈文字列〉］

です．このタイプは引数の組み合わせに使われ，

```
#define   MUL(A, B) (A)*(B)
```

は，

```
MUL(a+1, b+2)
```

の記述を

```
(a+1)*(b+2)
```

と解釈するよう働きます．

マクロ定義が次の行にわたるときは，“¥”を現在行の末尾に置きます．

なお，マクロ定義は

#undef 〈識別子〉

で同名の識別子が指定されるまで有効です．

●条件付きコンパイル

この機能は，そのときの都合に応じてプログラム記述ブロックをバイパスできるようにするためのものです．

制御対象の範囲は**#endif** までで，簡単なものでは

#ifdef 〈識別子〉 ………定義されていれば有効にする
#ifndef 〈識別子〉………定義されていなければ有効にする

があります．これらは，以前に#define で定義されているかどうか調べるものです．

#ifdef と同じようなことは

#if defined (〈識別子〉)

でも行なえます．

#if は普通

#if 〈定数式〉

の形態で使われ，たとえば

```
#define   DBGMODE 2 (デバッグ・モード２を指定)
```

のような定義のあとで，

```
#if    DBGMODE<1
```

などの条件判断が行なえます．その結果が真ならば続くブロックが有効となり，偽ならば無効となります．

これらの条件制御では，#endif以前ならば**#else**を使って結果が偽の場合に有効となるブロックを設定できます．また#ifでは，偽の場合#elseの前に#elifを必要な回数だけ記述できます．#elifの書き方は#ifと同じで，その結果も偽の場合は次の#elifまたは#elseに制御が移ります．

各条件制御はネスティングでき，たとえば次のような記述が可能です．

```
#ifdef    DBGMODE
    #if    DBGMODE==0
             ：（デバッグ・モード0のとき実行するブロック）
    #elif    DBGMODE==1
             ：（デバッグ・モード1のとき実行するブロック）
    #else
             ：（その他のとき実行するブロック）
    #endif(*1)
#else
    #define  NODBG(*2)
#endif(*3)
```

この場合，最初に出現する＊1のendifは，＃ifのためのもので，＊3は＃ifdefの締めくくりです．また，＊2のように＃defineをブロック内に含めることができます．

以上のプリプロセッサ制御命令は，コンパイル前にコンパイルすべきソース・プログラムの内容を最終的に確定するのに使われ，プリプロセッサがすんだらこれらははずされて，できあがったソースだけが以下のコンパイル処理に渡されます．

2-16 INCLUDEファイルの作り方

よく使うユーザ作成のサブルーチン(関数)などは，その都度入力するのは面倒でもあり，間違いも起きやすいので，INCLUDEファイルを利用して引用するのがベストです．

インクルード・ファイルはソース・プログラム形式になっているので，サブルーチンはほとんど原形のまま登録できます．「登録」といってもエディタで作成できるファイルであり，ユーザの常日頃利用しているディレクトリ内に置けるものですから，何も特別な手続きを必要としません．

図2.7は，整数値を標準出力に文字として表示する関数を登録したもので，ライブラリ名は“debug.h”

●図2．7　INCLUDEファイル（debug.h）の内容

```
#ifdef STDS
      #include <stdio.h>
      #include <stdlib.h>
#endif

void iprint (num, mod)
int num, mod;
{
    char s[10], *str=&s;
    itoa (num, str, mod);
    puts (str);
}
```

STDSが定義されているとき stdio.hとstdlib.hを参照する

整数値から進数記述(mod)に従って文字に変換し表示する

●図２．８　debug.hを参照するプログラム例（tt）

```
#define STDS              —debug.hのinclude記述を使う
#include "debug.h"        —debug.hを呼ぶ
main()
{
    int d=0x1234;
    iprint (d, 16);       —dを16進文字列で表示
    iprint (d, 10);       —dを10進文字列で表示
}
```

●図２．９　ttの実行結果

```
A>tt
1234    —16進値
4660    —10進値
```

です．ここではitoa関数で数値を文字列化し，puts関数でその結果を表示しています．たったこれだけのことですが，両関数の所属INCLUDEファイル名を調べたり，関数のパラメータの型などを吟味するとか，バッファを設定するといった雑多な手続きから開放されるので，デバッグ時には大いに役立ちます．#ifdef記述は，もしSTDSが定義されているとき，INCLUDEファイルの参照を肩代わりするためのものです．これによって，呼び出し側の負担を軽減できます．

参照する側のプログラムでは図2.8の例のように，カレント・ディレクトリ向きのINCLUDEファイル参照記述（"～"）を行ないます．このプログラム（tt）の実行結果は，図2.9のとおりです．

第3章

Cプログラミング入門と応用

3-1　表示(出力)することから始めよう

作ったプログラムが動くかどうかは，実際に走らせてみればわかります．とくに初めての言語で書いたプログラムが「通用する」かどうかは，走らせる以外に自力で知る方法はないでしょう．また，処理結果をみなければ正しく動いたかどうかの判定ができないので，そのためには出力関数を使わなければなりません．このため入門者は，第一に出力関数をマスターしてください．

そこで，手始めに適当な文字列を表示するプログラムに取り組んでみたいと思います．

文字列を出力するのに適した関数の代表的なものは，putsです．この関数は，標準出力に指定文字列を出力する際，文字列の最後で自動的に改行を行ないます．したがってファイルのオープン，クローズもいらないし，改行制御も不要です．

ここでサンプル・プログラムを図3.1に示します．

その中のメッセージの1つであるmsg2は，char型**配列**なのでauto領域で初期値を与えることができず，extern領域に置いています．一方，msg1はポインタのみ定義し，実行文で代入して実際のメッセージのアドレスをセットしています．

putsの引数はメッセージのポインタと規定されており，msg1はポインタ変数なので問題ありませんが，msg2の場合は仕様どおりにまじめに記述すると，

```
puts(&msg2)
```

としなければなりません．しかし，引数はポインタで渡すことがはっきりしているので，

```
puts(msg2)
```

と書いても同じように処理されます．

プログラミング言語は,「百の説明より一つの実例」が理解を助ける力になるものです.第1章で考え方,第2章で関数のアウトラインを述べたので,本章では実際のプログラム例を紹介して仕上げをしたいと思います.

せっかくプログラムを掲載するのであればできるだけ役に立つものをと考え,一部を除いて実用的なものを取り上げました.したがって,これらのプログラムをユーティリティ的な感覚で利用するのも1つの方法でしょう.

なお,アセンブラと関連のある部分は,読者がアセンブラについてひととおりの知識があるものとして進めています.そうでない場合は,第5部をマスターしてから読んでください.

なお,msg2はmsg2[0]と同じです.何度もいいますが,文字列は配列であることを忘れてはなりません.[0]は配列の先頭であり,言い換えると,文字列の先頭です.

msg1に代入するメッセージ内容の文字列は,始めからメモリ内のデータ領域中にとられ,代入時にそのアドレスがポインタ(msg1)に入れられるだけです.

図3.1のプログラムをテストした結果は,図3.2のとおりです.

●図3.1 メッセージ表示プログラム例(tr1)

```
#include <stdio.h>

extern char msg2[]="OK";
void main()
{
     char *msg1;
     msg1 = "Message display test";
     puts (msg1);
     puts (msg2);
}
```

●図3.2 tr1の実行結果

```
A>tr1
Message display test
OK
```

3-2 コマンド・パラメータの取り込み

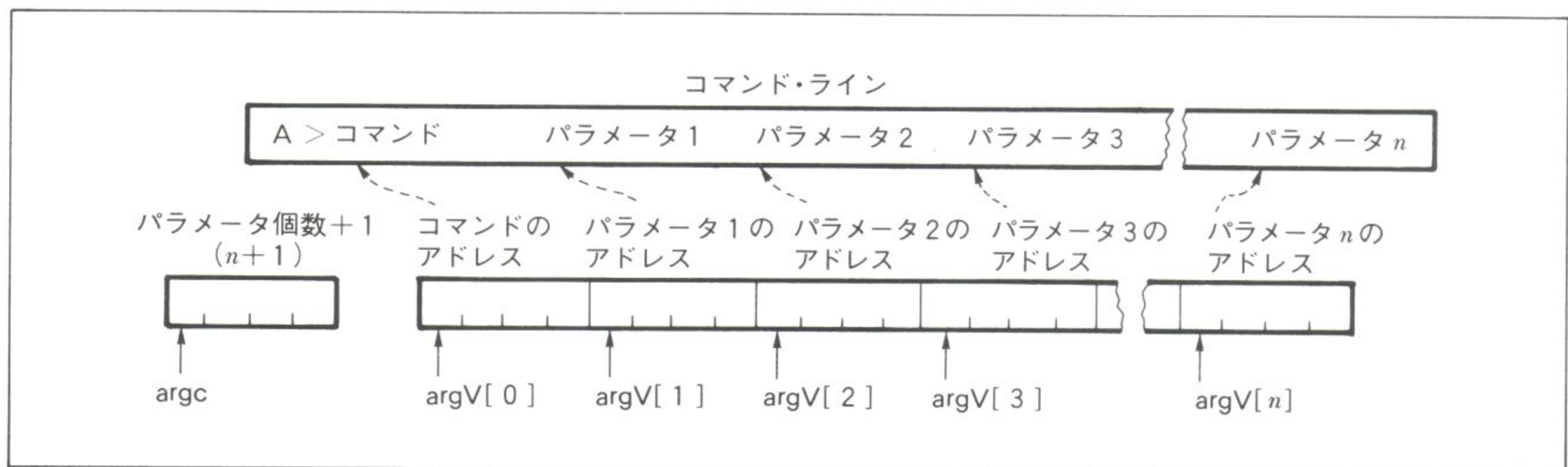
Cで開発したプログラムを起動する際，コマンド行にパラメータが指定できると非常に便利です．こうすることによって，操作時の入力を省けるだけでなく，バッチ・ファイルからの自動処理も可能になるからです．

コマンド・パラメータの取り込みには，main( )に渡される**argc**，**argv**［　］を利用します．これらは関数に渡されるパラメータと同じ性格をもつ変数で，argcにはコマンド・パラメータの個数，argv［　］には個々のコマンド・パラメータのアドレスが入っています．(図3.3)．

パラメータにはコマンド名(そのプログラム自身のファイル名)も含まれていますが，普通は参照しないので無視します．

参考プログラム(図3.4)では，コマンドに続くパラメータを，コマンド・ラインに記述された数だけ取り込んで出力するのですが，forループで1からargcの1つ手前まで繰り返すことで，実質的なパラメータ数に対応させています．パラメータの転送は，argv［i］が示すアドレスから行ない，iは実行後1つ先に進めさせます．こうすれば，次回に次のパラメータのアドレスが入っている位置から処理できることになります．

図3.4のプログラムの実行例を図3.5に示します．

●図3．3　main( )で受け取ることができるコマンド・パラメータ

●図3．4　コマンド・パラメータを1つずつ取り出して標準出力に転送するプログラム（ec）

```
/*
      Echo of Parameters
*/
#include <stdio.h>
main (argc, argv)
int   argc;
char *argv[];
{
      int   i;
      for (i=1; i<argc; i++) {
           puts (argv[i]);
      }
}
```

●図3．5　ecの実行結果

```
A>ec Par1 Par2 Par3 "Test string"
Par1
Par2
Par3
Test string
```

3-3 環境変数の取得

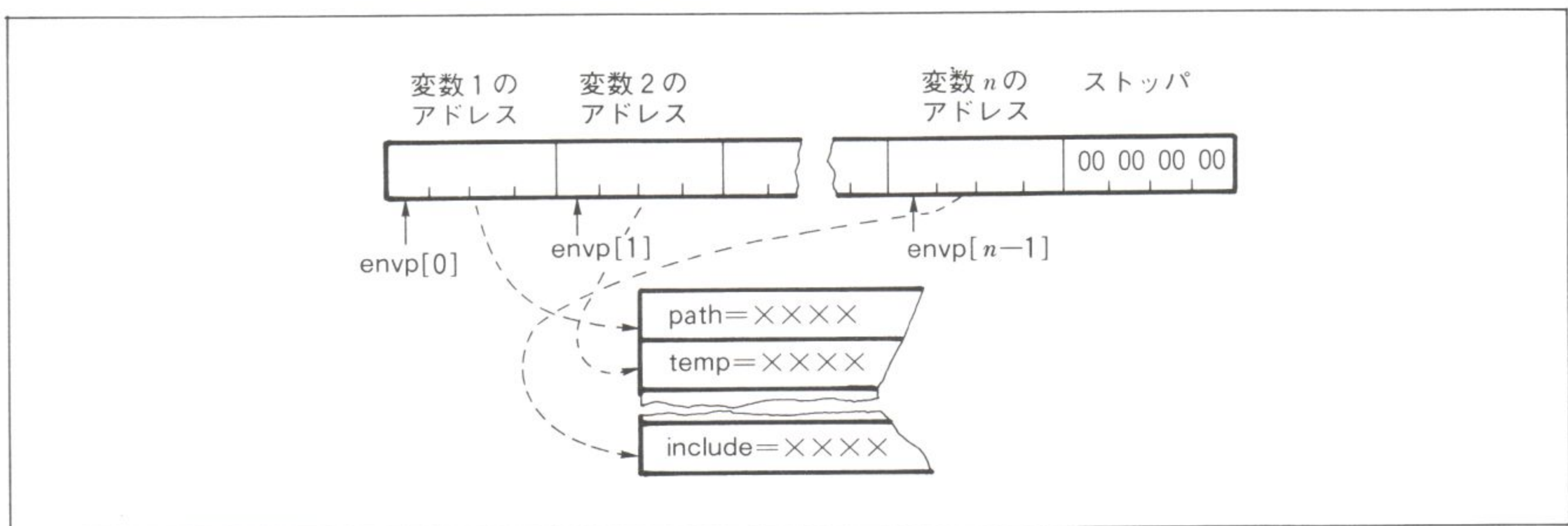
一般のプログラムで環境変数の内容を参照することはあまりありませんが，自作ユーティリティ・プログラムなどではその可能性があるので，参照するための方法を紹介しておきます．

環境変数を参照する手掛かりは，main( )に渡される引数の中のenvp[　]に入っています．この値は，各種の環境変数記述へのポインタで，テーブルの最終項目の次はストッパ(アドレス値＝０)が付いています(図3.6)．したがって，テーブルからの取り出しはアドレス値のゼロを検出するまでとなり，前もって個数はわかりません．

このような構造を利用して，環境変数を取り出して標準出力に出力するプログラムを図3.7に示します．ここで，mainのパラメータとして関係のないargc，argvも含まれている点に注意する必要があります．これらは参照しなくてもこの順序で与えられるので，一応書いておかないとenvpの位置がズレてしまうことになります．

このプログラムを実行した結果は，図3.8のとおりです．

同じ内容はgetenv( )関数で取得でき，また

●図３．６　mainで受け取ることができる環境変数

●図３．７　環境変数の内容を出力するプログラム（env）

```
/*
     Display of Environment Variables
*/
#include <stdio.h>
main (argc, argv, envp)
int   argc;
char *argv[];
char *envp[];
{
      int    i;
      for (i=0; envp[i]>0; i++) {
           puts (envp[i]);
      }
}
```

●図３．８　envの実行結果

```
A>env
path=A:¥;A:¥SYS;A:¥BIN;A:¥CC;A:¥BC;A:¥BASIC2;¥MyProgs
temp=C:
lib=a:¥lib
include=a:¥include
```

set ↵

コマンドによっても出力されるので，参考までに付け加えます．ここでは，ポインタを使って配列の内容を参照する手法を説明したかったので，上記のような方法としました．前節の方法と比べてみると面白いと思います．

3-4 メモリ・ダンプ関数の作り方

各言語を使ったサンプル・プログラムとして，メモリ・ダンプがよく題材になります．これは，プログラムの規模がさほど大きくもなく，アドレスによってデータを参照するとか，16進変換，文字出力など比較的多方面にわたる処理を必要とするので，格好の教材として利用できるからです．ここではそういった目的を兼ねて，できるだけ広範囲にサンプルを提供したいと思います．

テストをしやすくするため，プログラムは最初，main( )を含めた形で作ります(図3.9)．mydump関数は戻り値が不要なvoid型なのですが，まじめに定義するとmain( )の前に置かなければならなくなるので，手抜きをして型名無宣言(int型となる)にしておきます．

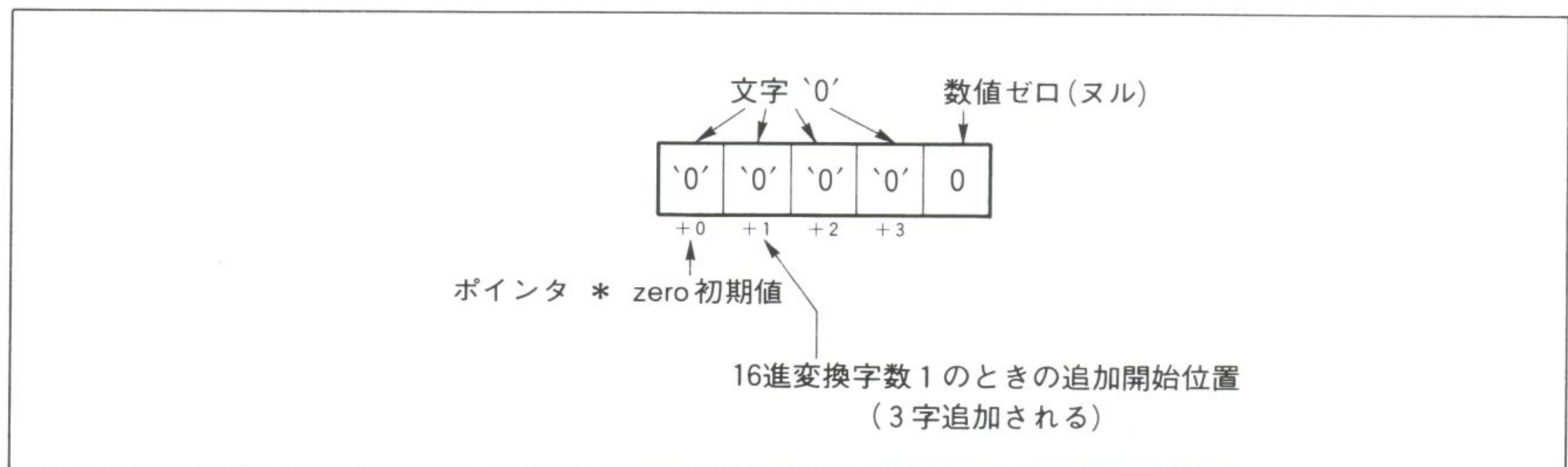
ダンプ出力の行当たりの内容は，最初にその行に表示するメモリの先頭アドレス，続いてデータの16進4桁表示を8ワード分，同じデータの文字表示16バイト分と内容が固定されています．この形をくずさないほうがプログラムを簡単にできるので，ここではダンプ終了アドレスの判定は，1行分の出力がすんだ時点で行なっています．

プログラムで目につくのは，dump関数で共用体を利用していることです．これは，16進表示するために読み出した2バイト・データ(ショート型整数)を，1バイトずつに分割したchar型としても利用できるようにするものです．

1行の表示内容は，バッファlnに作成します．プログラムではアドレス──→文字列変換(itoa)の結果で最初のlnを作り，これをベースにstrcatで途中のスペースやデータのダンプ結果をどんどん追加していきます．この過程で文字表示の内容は，バッファclに別途こしらえておき，16バイト分の処理が終わるごとにlnに連結して表示したところで1行分の処理を終わります．

問題は16進文字変換するuwtoaなどの関数がサービス過剰で，頭のゼロ・サプレス(ゼロを消すこと)までやって，さらにその分左詰めにしてしまうことです．ことダンプに関してはちょっとやり過ぎというべきで，このおかげで欠けたゼロを補う処理が必要になります．そのための方法として，このプログラムでは4字のゼロ文字(zero)の定数をとり，16進変換後の文字列の字数を差し引いた字数だけ付加するようにしています．このzeroはポインタですから，ゼロ文字列から任意の字数を切り取るのは簡単です(図3.10)．

プログラムの実行結果は，図3.11のとおりです．

●図3．10　4字のゼロ文字定数とポインタ

●図3．9　ダンプ・プログラムの原形（mdump）

```
#include <stdio.h>
#include <stdlib.h>

main()
{
    int fr=0x87100, t=0x871ff;
    mdump (fr, t);
}

mdump (from, to)
int from, to;
{
    unsigned short int *dt, i, sz;
    char ln[80], ss[5], cl[16], *zero;
    union {
        unsigned short int dd;
        char cc[1];
    } un;
    for (dt=from; dt<to; ) {
        itoa (dt, ln, 16);
        cl[0] = 0;
        strcat (ln, ": ");
        for (i=0; i<16; ) {
            strcat (ln, " ");
            un.dd = *dt++;
            uwtoa (un.dd, ss, 16);
            sz = strlen (ss);
            zero = "0000";
            zero += sz;
            strcat (ln, zero);
            strcat (ln, ss);
            cl[i++] = iscn (un.cc[0]);
            cl[i++] = iscn (un.cc[1]);
        }
        cl[16]=0;
        strcat (ln,"  ");
        strcat (ln, cl);
        puts (ln);
    }
}

iscn (x)
unsigned char x;
{
    int r;
    if (x >= 0xE0)
        return ' ';
    else
        if (x < 0x80)
            {
            r = iscntrl(x);
            if (r == 0)
                return x;
            else
                return '.';
            }
        else
            {
            if (x < 0xA0)
                return ' ';
            else
                return x;
            }
}
```

●図３．11　mdump の実行結果例

```
A>mdump
87100:  0000 0000 6162 6F72 7400 C001 0000 003A  ....abort.ﾀ.....:
87110:  7465 7874 0000 C002 0000 0000 6461 7461  text..ﾀ.....data
87120:  0000 C003 0000 0000 6273 7300 C004 0000  ..ﾀ.....bss.ﾀ...
87130:  0000 7374 6163 6B00 B201 0000 0000 5F61  ..stack.ｲ......_a
87140:  626F 7274 0009 B2FF 0000 0001 5F5F 696F  bort..ｲ ....__io
87150:  6200 B2FF 0000 0002 5F66 7072 696E 7466  b.ｲ ...._fprintf
87160:  0010 B2FF 0000 0003 5F5F 6578 6974 0066  ..ｲ ....__exit.f
87170:  2001 0000 0000 1001 4879 4601 0000 001C   .......HyF.....
87180:  1001 4879 56FF 0001 0000 002C 1001 4EB9  ..HyV ......,..Nｹ
87190:  46FF 0002 1005 3EBC 0003 4EF9 46FF 0003  F ....>ｼ..N F ..
871A0:  101D 4162 6E6F 726D 616C 2070 726F 6772  ..Abnormal progr
871B0:  616D 2074 6572 6D69 6E61 7469 6F6E 0A00  am termination..
871C0:  0000 4142 532E 4F00 0000 0000 0000 0000  ..ABS.O.........
871D0:  0000 0000 0000 0000 0000 0000 007C 9254  .............| T
871E0:  10A9 D000 0000 0000 6162 7300 C001 0000  .ｩﾐ.....abs.ﾀ...
871F0:  0016 7465 7874 0078 C002 0000 0000 6461  ..text.xﾀ.....da
```

3-5　メモリ・ダンプ関数のライブラリへの登録

ここでは完成したメモリ・ダンプ・プログラムのうち，テスト用に作成した main( )を除いた本体をライブラリに登録する手続きについて説明します．

エディタで main( )部分を除去したあとは，ダンプ関数の定義は本来の

```
void  mdump(from, to)
```

に戻します．main( )をはずしてしまえば，mdump( )はそのまま void を指定してもエラーにはなりません．戻り値がないことをはっきり宣言するためにも，正しく定義しておくのが適当です．このプログラムは，

```
cc  mdump.c  ／L
```

によってひとまず mdump. o ファイル止まりでコンパイルしておきます．

さて，main( )部もこれで用ずみになった訳ではなく，まだ使い途が残っています．それは，mdump 関数切り離し後のリンク・テスト用としての活用です．今後 mdump 関数をリンクするためには，図3.12のように外部宣言をして，リンカで呼び込む手続きをしなければなりません．この手続き記述が適当かどうかは，main( )を残しておくことによってただちにテストできます．この部分は一応“md_main. c”として

```
cc  md_main.c  ／L
```

により md_main. oファイルを作ります．

次にリンカを使い，

●図３．12　mdump のメイン部を切り離してライブラリ・テスト用にしたもの

```
void mdump();

main()
{
     int fr=0x87100, t=0x871ff;
     mdump (fr, t);
}
```

```
lk md_main mdump.o ¥lib¥clib.a
```

で実行形式のmd_main.xを作ります．これを走らせてうまくいけば，テスト用のmain( )も，mdump( )も完成です．

完成ずみのmdump( )はこのままにしておいても使えないことはありませんが，Cコンパイラ・ドライバを使う際にはアーカイブ・ファイルに入れておかないと使えません．それもclib.aに入れておかないと面倒です．clib.aからユーザーが常に使用するものを抜き出し，ユーザー専用のアーカイブ・ファイルを作るとしても，それ以外の関数は犠牲になるので，必要なものはその都度追加しなければならず，手数上も，スペースの点から見ても不経済です．

clib.aへの登録は，次のコマンドで行なえます．

```
ar clib mdump.o (mdump.oが¥lib内にある場合)
```

念のため，このあと

```
ar clib /L
```

でclib.aの内容を点検しておくことが望まれます．というのはアーカイバは更新時に別途出力用ファイルを作り，スペースが足りなくなると何のメッセージも出さずに終了してしまうからです．その結果，既存の関数まで失われるとか，clib.aそのものが使えなくなるといった問題が生じます．このことは，Ver.1.01現在で解決されていません．アーカイバを使うときは，バック・アップの用意を忘れないようにしましょう．

アーカイブ・ファイルに登録できたら，次はmd_main.cをCコンパイラ・ドライバによってコンパイルし，実行します．これでうまく動作できたら，md_main.cや，md_main.xなどは消去してもかまいません．

次のステップは，必要ならばmdump( )関数を呼び出す外部定義をINCLUDEファイルに登録することです．これはmd_main.cのmain( )以前をそのまま利用すればよく，しかもたった1行だけなので話は簡単です．登録先は，前章で作成したdebug.hで，このファイルのあとに追加するだけです．

以上でユーザ作成関数が，標準関数と同様に使えるようになったのですが，上述のプロセスはあくまで一歩一歩確認しながら進める方法で，初心者向けの手順を示したにすぎません．慣れてきたら，いきなりゴールを目指したやり方もできるようになりますが，プログラムに虫はつきものなので，最小限前節のようにmain( )とユーザ関数をひとつのソース・ファイルに収容したテストぐらいはしておきたいものです．そうしないと，時間のかかるアーカイバを何度も実行しなければならなくなってしまいます．

3-6 アセンブラ変換について

Cコンパイラは，C言語により記述されたプログラムを，直接機械語に変換する訳ではありません．Cコンパイラの処理範囲はアセンブラに変換するところまでで，あとはアセンブラとリンカが仕上げを行ないます．

したがって，中間結果であるアセンブラのソース・プログラムは最終的には不要になるものですが，コンパイルの結果を調べるときはきわめて大きな手掛かりを与えてくれます．

図3.13は，本章の始めに紹介した図3.1のCプログラムのアセンブラ・ソース変換結果です．

ここで，_msg2はCプログラムのmsg2に相当するもので，16進で定義されていますが，内容は単に文字コードに置き換えただけでまったく同じです．

_mainはCプログラムのmainに相当し，実行に際しL2で必要なスタック領域(auto領域が使用)を確保しています．この場合＊msg1のポインタだけなので，4バイトとられます．

●図3．13　図3．1のプログラムのアセンブル変換結果

```
        include fefunc.h
*
*
*_msg2
*
*
        .GLOBL  _msg2
        .DATA
_msg2:
        .DC.B   $4f,$4b,$00
        .EVEN
*
*
*_main
*
*
        .XREF   __main
        .XDEF   _main
        .TEXT
_main:
__main:
        BRA     L2
L3:
        MOVE.L  #L5,-4(A6)
        MOVE.L  -4(A6),-(SP)
        JSR     _puts
        ADDQ.L  #4,SP
        MOVE.L  #_msg2,-(SP)
        JSR     _puts
        ADDQ.L  #4,SP
L4:
        UNLK    A6
        RTS
L2:
        LINK    A6,#-4
        BRA     L3

*
*
*STRING AREA
*
*
        .DATA
L5:
        .DC.B   $4d,$65,$73,$73,$61,$67,$65,$20,$64,$69,$73,$70,$6c,$61
        .DC.B   $79,$20,$74,$65,$73,$74,$00
        .EVEN
        .END
```

関数の実行に際しては，引数をスタックに上積みして，該当サブルーチンにJSRでジャンプし，戻ってからスタックを開放する手順を踏んでいます．このとき，msg1の側では事前に代入文の実行により＊msg1の値をセットします．auto級変数の処理にA6相対アドレス（正式には「ディスプレースメント付きアドレス・レジスタ間接」）が使われ，＊msg1に転送する値としてL5（msg2に代入する文字列）のアドレスが用いられていることが読みとれれば，あとの説明はいらないでしょう．

msg1の側では，引数のアドレス値（ポインタ）はスタック（auto領域変数）から与えられますが，msg2側では直接そのアドレスが引き渡されています．

Cのリストと見比べてみると，細かなニュアンスの違いが表現されていることがよくわかると思います．

3-7 アセンブラ・プログラムの挿入

Cコンパイラがアセンブラ・ソースを生成することは，すでに述べたとおりですが，ソースの一部を生の

●図3．14　tr2.cのリスト

```
#include <stdio.h>

void main()
{
     char *msg1;
     msg1 = "Message display test";
     puts (msg1);

#asm
     move.l #asm_msg,-(sp)
     jsr     _puts
     addq.l #4,sp
#endasm

}

#asm
     .data
asm_msg:
     .dc.b "OK from ASSEMBLER program",0
#endasm
```

（#asm〜#endasm の1つ目：アセンブラ挿入部1、2つ目：アセンブラ挿入部2）

●図3．15　図3．14のプログラムをアセンブラに変換した結果

```
 include fefunc.h
 *
 *
 *_main
 *
 *
         .XREF    __main
         .XDEF    _main
         .TEXT
_main:
__main:
         BRA      L1
L2:
         MOVE.L   #L4,-4(A6)
         MOVE.L   -4(A6),-(SP)
         JSR      _puts
         ADDQ.L   #4,SP

     move.l #asm_msg,-(sp)
     jsr     _puts
     addq.l #4,sp
L3:
         UNLK     A6
         RTS
L1:
         LINK     A6,#-4
         BRA      L2


     .data
asm_msg:
     .dc.b "OK from ASSEMBLER program",0
 *
 *
 *STRING AREA
 *
 *
         .DATA
L4:
         .DC.B    $4d,$65,$73,$73,$61,$67,$65,$20,$64,$69,$73,$70,$6c,$61
         .DC.B    $79,$20,$74,$65,$73,$74,$00
         .EVEN
         .END
```

（move.l〜addq.l：アセンブラ挿入部1、.data〜.dc.b：アセンブラ挿入部2）

●図3．16　tr2のテスト結果

```
A>tr2
Message display test
OK from ASSEMBLER program
```

まま送り出す機能も提供されています．

このための制御文は，

```
#asm
  ∫
#endasm
```

で，間に書かれた文はそのままアセンブラに渡されます．この部分を中心にしてCを見れば，Cが「高級アセンブラ」と呼ばれる意味がよくわかります．

この機能を利用する際は，前節で述べたアセンブラ・ソースへの展開のイメージをよく呑み込んでおく必要があり，挿入部分がちぐはぐな場合は，実行の段階でうまくいかないので注意しなければなりません．

図3.14は前節で紹介したプログラムのmsg2をアセンブラで定義し，その表示もアセンブラで行なうように変更したものです．参考までにこのプログラムをCコンパイルして生成されたアセンブラ・ソースを図3.15に示します．このようにしてみると，asm_msgを定義するデータ領域を別途#asm～#endasmの中に記述した理由がわかると思います．

このプログラムの実行結果は図3.16のとおりです．

3-8 BASICプログラムのC展開

XCではBASICプログラムもCに変換できます．ということは，そのリストを参照することが可能なので，ここではテスト・プログラム(図3.17)を用いて展開させてみました．

●図3．17　テスト用のBASICプログラム（tr3.bas）

```
10 str a="PRINT TEST from BASIC program"
20 str dummy
30 print a
40 input dummy
50 end
```

●図3．18　tr3.basをC変換した結果（tr3.c）

```
#include        "basic0.h"
static  unsigned char   a[32+1];
static  unsigned char   dummy[32+1];

/********** program start **********/
void
main(b_argc,b_argv)
int     b_argc;
char    *b_argv[];
{
b_init();
 b_strncpy(sizeof(a),a,"PRINT TEST from BASIC program");
 b_sprint(a);b_sprint(STRCRLF);
 b_input("? ",sizeof(dummy),dummy,-1);
 b_exit(0);
}

/****************************/
```

●図3．19　tr3 の実行結果

```
A>tr3
PRINT TEST from BASIC program
?
```

結果は図3.18のとおりで，文字列 a の初期値設定が main( )の中で行なわれていることなどがわかります．このサンプルでは，以下がＣの文 BASIC の命令に１対１で対応しています．

原プログラムでダミーの input 命令を使っているのは，BASIC プログラム終了時にスクリーンがクリアされて結果が見れないので，一時停止させるためです．プログラムを実行させて，“？”の表示に対し，↵を押すとプログラムが終了します．

このプログラムの実行結果は図3.19のとおりです．

3-9 BASICにもCプログラムが挿入できる

Ｃプログラムにアセンブラ・プログラムを挿入できたように，X-BASIC ではＣ変換する BASIC プログラム中にＣプログラムを挿入できます．

制御文は，

```
#c
 ∫
#endc
```

で，間にＣプログラムを書きます．

図3.20は，前節のプログラムにＣプログラムによるメッセージ表示を追加したもので，これをＣ展開すると図3.21のようになります．

リストで比較してみると，BASIC のメッセージ表示とＣのそれとでは使用関数が異なるのがわかります．Ｃの方がたくさんの関数が使えるので，Ｃプログラムの挿入により，プログラム全体の制約事項を少なくすることができます．同じことは，Ｃプログラムにアセンブラ・プログラムを挿入する場合にも当てはまります．

BASIC の欠点は，print 命令がスクリーンのみに限定され，プリンタやディスクにリダイレクトできないことです．しかし，挿入Ｃプログラムで puts( )を使えば，この問題は解決されます．

参考までにつけ加えると，Ｃソース，アセンブラ・ソースともに，ファイルとして作成されて残るので，直接エディタで書き加えるという挿入方法もあります．ただし，この方法は後日リストを読む際にわかりにくくなる欠点があるので，とくに必要のない限り採用しないほうがよいと思います．

●図3．20　Ｃコーディングを含む BASIC プログラム（tr4.bas）

```
 10 str a="PRINT TEST from BASIC program"
 20 str dummy
 30 print a
 40 cprint()
 50 input dummy
 60 end
 70 func cprint()
 80 #c
 90     char *msg;                           ┐
100     msg = "PRINT TEST from C subroutine";  │ C挿入部
110     puts(msg);                           ┘
120 #endc
130 endfunc
```

●図3．21　tr4.basをC変換した結果（tr4.c）

```
#include        "basic0.h"
static  unsigned char   a[32+1];
static  unsigned char   dummy[32+1];
int     cprint();

/********** program start **********/
void
main(b_argc,b_argv)
int     b_argc;
char    *b_argv[];
{
b_init();
 b_strncpy(sizeof(a),a,"PRINT TEST from BASIC program");
 b_sprint(a);b_sprint(STRCRLF);
 cprint();
 b_input("? ",sizeof(dummy),dummy,-1);
 b_exit(0);
}

/****************************/
int     cprint()
{
     char *msg;                                  ┐
     msg = "PRINT TEST from C subroutine";       │ C挿入部
     puts(msg);                                  ┘
 }

/****************************/
```

●図3．22　tr4の実行結果

```
A>tr4
PRINT TEST from BASIC program
PRINT TEST from C subroutine
?
```

BASICプログラムにCプログラムを挿入し，その中でアセンブラ・プログラムを挿入するという「入れ子」も可能です．

なお，このプログラムでputsの出力をディスクにリダイレクトするには，前もってそのファイルを作っておく（ダミーの内容を書いておく）必要があります．なぜならここでは，クリエイトする命令を使っていないからです．

Cプログラム挿入時の注意点としては，BASICプログラムのfuncによるサブルーチンの内部またはあとに挿入部が記述されていないとき，それがたとえendのあとにあっても，main( )の中に入れられてしまいます．ここではその問題を避けるために，BASIC側でサブルーチン（関数）の枠を作っています．BASICのサブルーチンが1つもないときは，この点についてとくに意識してかからなければなりません．

tr4の実行結果は図3.22のとおりです．

第5部

アセンブラ・プログラミング

第1章

アセンブラの基礎

1-1　アセンブラ・プログラムの開発手順

BASICでは，インタプリタによってソース・プログラムを入力するだけでプログラムが実行できます．Cではcc(Cコンパイラ・ドライバ)を使えば，自動的に実行形式のプログラム・ファイルができあがります．これらの言語を使う限りは，実行形式のプログラム・ファイルについて意識することは少なく，ましてリンカの存在を考えることはほとんどないでしょう．

アセンブラのプログラムを開発する際には，図1.1のように，単にアセンブラを起動すればすむということはありません．アセンブラはあくまで中間ファイルにすぎない**オブジェクト・ファイル**を作るだけで，**リンカ**を経てようやく実行形式のプログラム・ファイルができあがります．これは分割作成したプログラムや，他言語で作成したプログラムなどとのリンクのために2段階に分離しているためで，16ビット機以上のパソコンでは，このような構成が常識になっています．

したがって，アセンブラ・プログラムを開発するときは，そのプログラム・モジュール内部の整合性だけでなく，結合するモジュールについても外部参照時の整合性をもたせなければなりません．

初めてのプログラミングのときは，リンカの機能はあまり利用せず，できるだけモジュール一本だけで完結するようにすれば失敗しなくてすみます．当分の間リンカはほとんど通過するだけとし，充分に慣れてから本来の機能を使うようにするのがよいでしょう．

アセンブラの起動は，

as　[{<スイッチ>}] <ソース・ファイル名>

で行ないます．

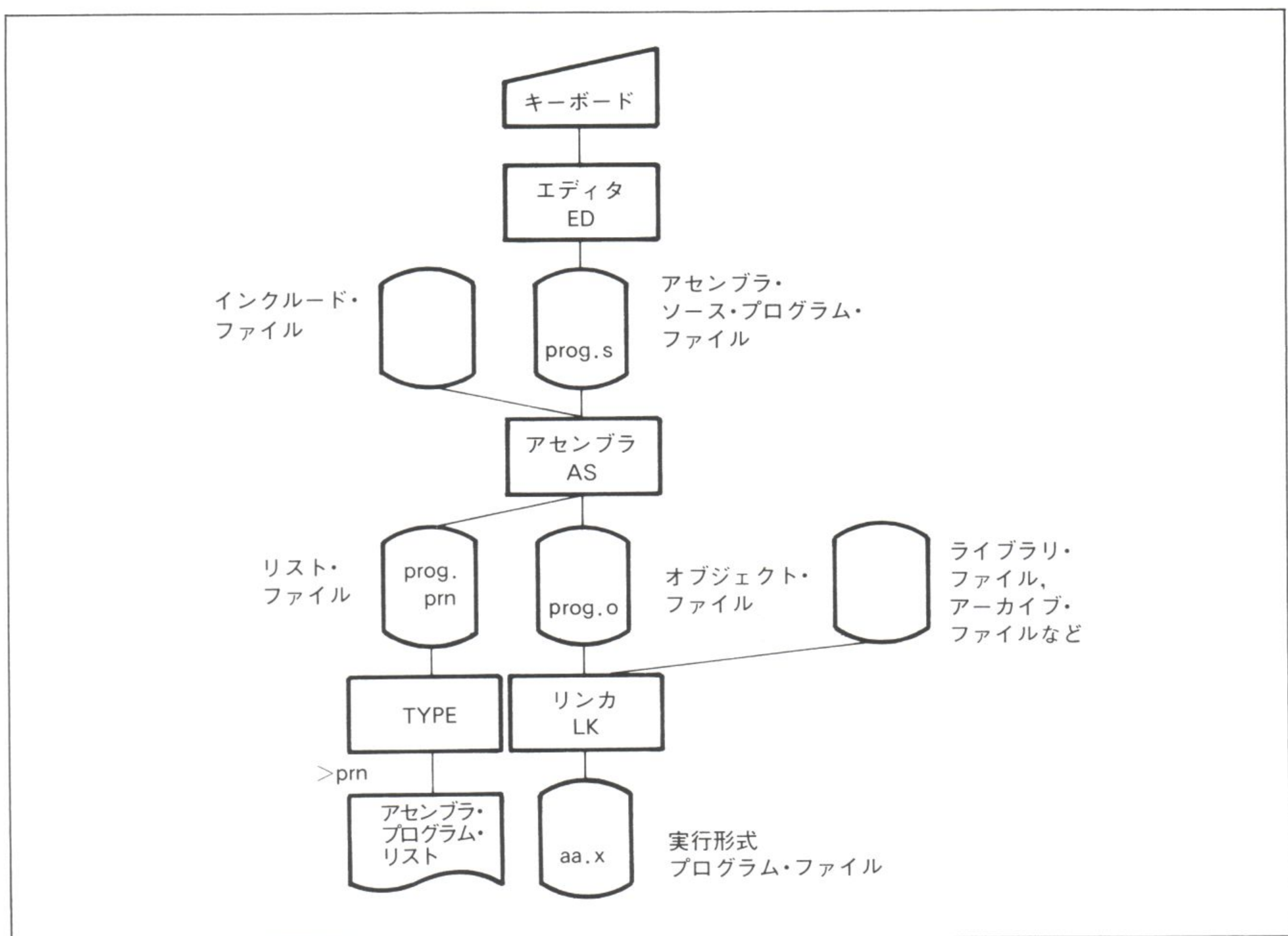
スイッチの内容は，表1.1のとおりで，／O を指定しなければオブジェクト・ファイルが出力されません．たいがい最初の数回までのアセンブル時にはエラーが出て，オブジェクト・ファイルを作成しても無意味

X68000プログラミングの行きつくところは，結局アセンブラです．BASICでもCでも満足できなければ，アセンブラを使うしかありません．

しかしアセンブラは機械語と直接対応するだけに，他のどの言語でもできなかった処理が可能になる反面，同じことをするにもたくさんの命令を使わなければなりません．この点をカバーするために，マクロを使ったり，ライブラリにサブルーチンなどを蓄積する工夫がなされています．

最も賢い方法は，プログラムの大半をBASICやCで書き，必要な部分だけにアセンブラを使う方法です．こうすれば，後日修正するときもわかりやすく，開発の手数も少なくてすみます．

プログラムのサンプルは，付録のOS-9の説明のところにも掲載されているので，参考にしてください．

●図1．1　アセンブラ・プログラムの開発手順

●表1．1　アセンブラのスイッチ一覧表

スイッチの記述	意　　味
/T　〈パス名〉	テンポラリ・ファイルのパス指定
/O　[〈ファイル名〉]	オブジェクト・ファイル名の指定
/I　〈パス名〉	インクルード・ファイルのパス指定
/P　[〈ファイル名〉]	アセンブル・リスト出力ファイルの指定
/N	最適化の抑止(ショート・ブランチ)
/W	警告メッセージの出力抑止
/U	未定義シンボルの外部参照指定
/D	全シンボルの外部エントリ指定
/8	シンボルの有効データ長の指定
/M　〈nn〉	シンボルの最大個数の指定(201～65,536)
/S　〈シンボル名〉	シンボルの定義

●表1．2　リンカのスイッチ一覧表

スイッチの記述	意　　味
/N　〈nn〉	シンボルの最大個数の指定(201～65,536)
/T　〈パス名〉	テンポラリ・ファイルのパス指定
/O　[〈ファイル名〉]	オブジェクト・ファイル名の指定
/B　〈ベース・アドレス〉	ベース・アドレスの指定
/I　〈ファイル名〉	インダイレクト・ファイルの指定
/V	バーボーズ・モード(リンク詳細情報表示)
/X	シンボル・テーブル出力禁止

なことが多いので，／Pでリストを確認してOKならば／Oを指定するようにします．／Pスイッチでファイル名を省略すると，ソース・ファイル名のあとに.prmを付けたファイルがとられます．／Oスイッチでは，ファイル名を省略すると，ソース・ファイル名に.oの拡張子を付けたものになります．しかし，

```
as ／o prog
```

のように指定すると，エラーになってしまう(progがオブジェクト・ファイル名とみなされ，ソース・ファイル名がなくなる)ので，最後のスイッチのファイル名は省略できません．なお，ソース・ファイル名の.sは省略できます．

また，リンカの起動は，

lk　[{<スイッチ>}] <メイン・オブジェクト・ファイル名> [{<オブジェクト・ファイル名>}]

で行ない，このとき表1.2のスイッチを使います．

スイッチの中でよく使われるのは／Oで，ここでファイル名を省略するとメイン・オブジェクト・ファイル名に拡張子.Xを付けたものになります．また，オブジェクト・ファイル名(メインのものも含む)の拡張子が.oのときは，拡張子は省略できます．オブジェクト・ファイルとして，ライブラリ・ファイルを使用することも可能です．

／Xのシンボル・テーブル出力禁止スイッチは，実行形式プログラム・ファイルに，シンボリック・デバッガで使用するシンボル・テーブルを出力しないようにするためのもので，デバッグが完了したプログラムをリンクするときに使います．これによりファイル容量，実行時メモリの節減が図れます．

リンク時の詳細な情報を知りたいときは，／Vスイッチを指定します．

1-2 データ長の記述(バイト，ワード，ロング・ワード)

68000の命令は，一般に複数のデータ長(図1.2)が扱えるようになっています．このため，命令を記述する際には，どのタイプのデータ長で実行するのかを明確にしておく必要があります．データ長が暗黙のうちに特定化されている命令についても，そのサイズを意識すべきなのはいうまでもありません．

68000で犯しやすいプログラム・ミスの典型的なパターンの1つは，データ領域でのデータ長の定義と，命令における実行サイズのくい違いです．

すなわちデータ領域におけるデータ長の定義は，dc(定数または初期値をもつ変数)，ds(変数)などのあとに，

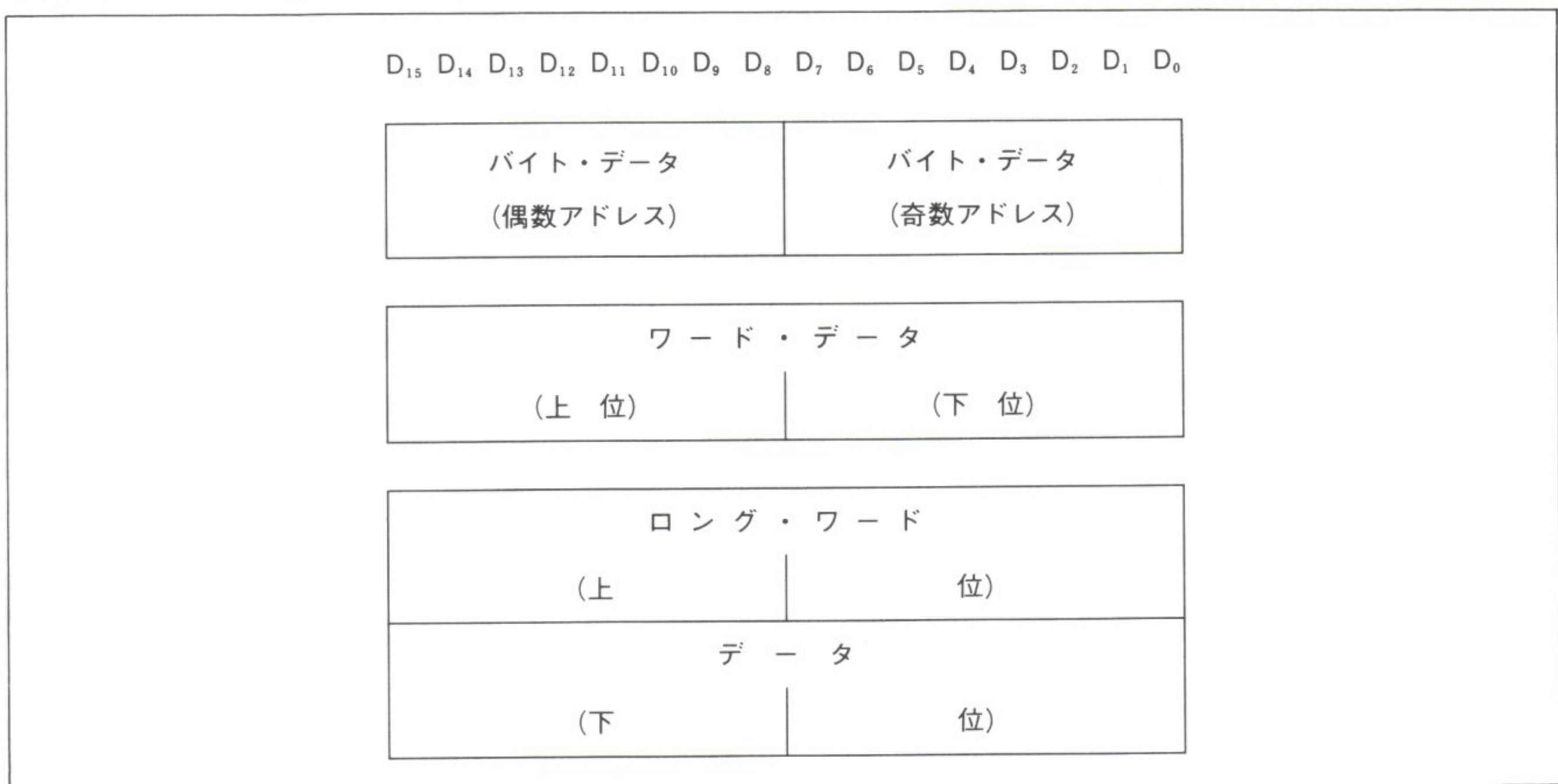
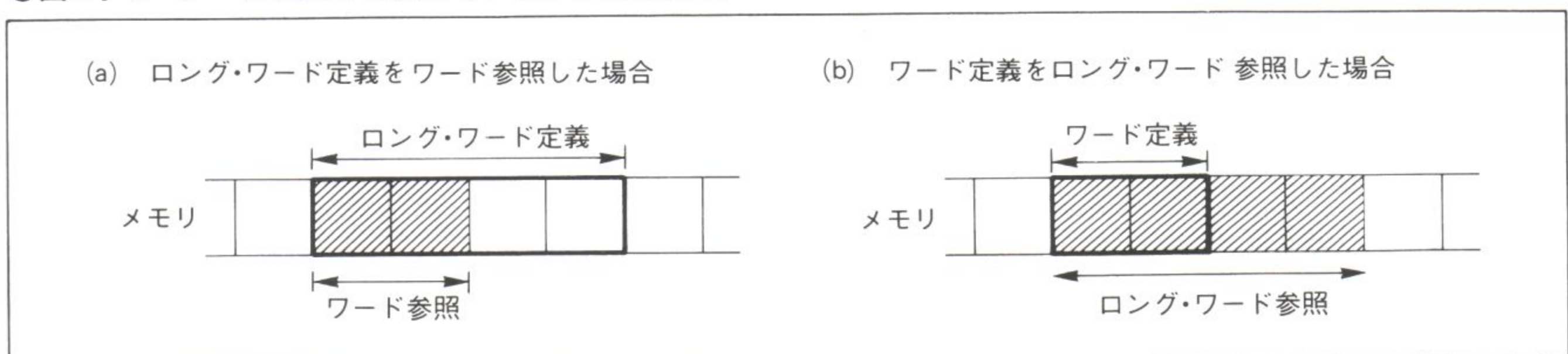
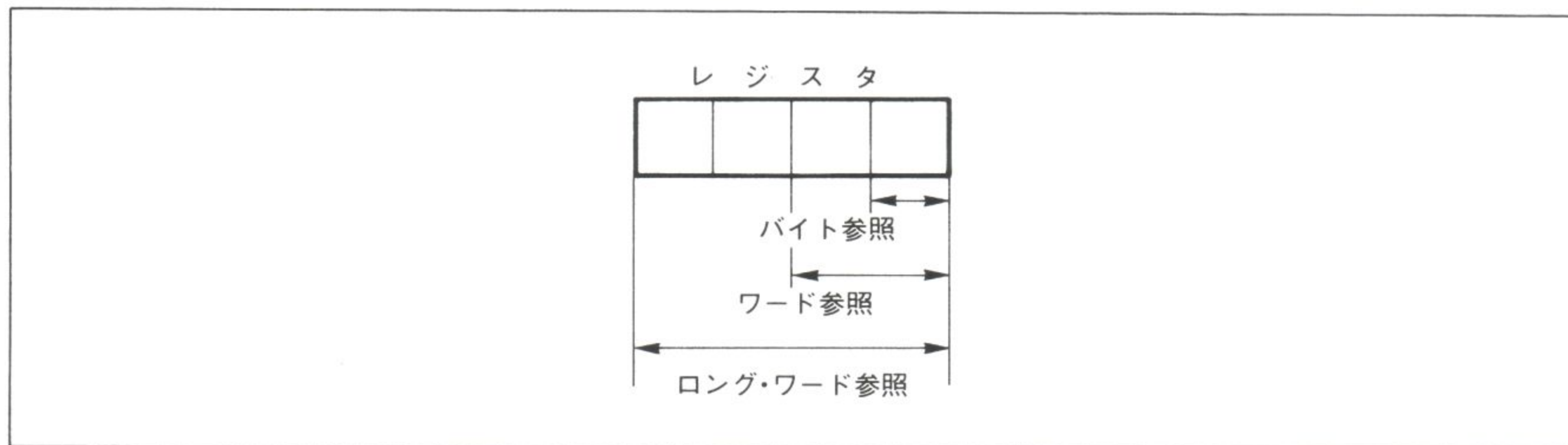
.l ……**ロング・ワード**（4バイト）
.w ……**ワード**（2バイト）
.b ……**バイト**（1バイト）

を付けることによって行ないます．例外として，文字列は複数バイトであっても.bで定義されます．命令の実行サイズについても同様で，

```
move.w
```

のように，命令のあとに付加する形をとります．

もしデータ領域での定義がロング・ワードで，命令の実行サイズがワードだったとすれば，実際に命令によって参照されるのはデータの上位ワード部分のみで，下位ワードは無視されます(図1.3(a))．

●図1．2　68000で扱えるデータ長の種類

●図1．3　データ定義と参照のくい違いによる問題

●図1．4　レジスタの参照サイズと参照される範囲

また，データ領域の定義がワードで，命令の実行長がロング・ワードだった場合は，書き込みを伴う命令では，後続のデータまで破壊することになってしまいます(図1.3(b))．

いずれの場合もプログラムは正常に動作せず，時として原因がなかなかわからないことがあります．一見して定義が一致しているようでも，よくよく調べてみると関連する命令のうち1命令だけ実行長が違っているということもあるのです．

実行長についてもう1つ知っておきたいことは，レジスタの場合は下位から数えたバイト数の範囲が参照されるという点です(図1.4)．したがって，同じレジスタの内容をメモリに収容する場合でも，実行長が異なれば収容先のラベルのアドレス位置の収容結果は異なります．

68000では，ワードとロング・データは必ず偶数アドレスから始まっている必要があります．このため，バイト・データを定義したあとにこれらのデータを定義するときは，必要に応じて

```
.even
```

擬似命令を使って，強制的に偶数アドレスに合わせることが行なわれます．

1-3 シンボル値の定義

❖関連コマンド　　equ, set, reg

アセンブラでは，リストをわかりやすくするため，定数などを直接オペランドに記述せず，別途ラベルを付けて定義した上で参照する形がとられます．このときの定義は，単にデータ値と対応づけするのみで，データ長は与えられません．もちろんこれは，アセンブル時だけの便宜的な手段にすぎないので，直接にデータ・エリアが生成されるといったこともありません．

シンボル値を定義する際に，最もよく利用されるのは **equ** です．この擬似命令は，たとえば，

```
cpu equ 68000
```

のように使用します．このとき，アセンブラは“cpu”という名前が出現するたびに68000を代入(置き換え)して処理することになります．equ は同一ラベルについて一度しか使えませんが，1つのプログラム・モジュール内全域にわたって有効です．なお，equ のオペランドに他のラベルを参照するような記述もできますが，前方参照(あとの命令の定義を参照すること)は許されていません．あくまで事前に定義されていることが必要です．

参照値をプログラムの部分によって変えたいときは，**set** 擬似命令を使います．この擬似命令の定義は，次の同ラベルの set 定義まで有効です．たとえば，

```
count   set 3   ························①
            ⋮
        move.b  #count, d0   ····②
            ⋮
count   set 4   ························③
            ⋮
        move.b  #count, d0   ····④
            ⋮
```

のとき，②では3が，④では4がd0に送られます．このケースでは，もし①以前にcountが参照されると，最初の定義値(①)がとられます．

これらの擬似命令では文字定数も定義できますが，文字コードが16進の数値に置き換えられます．したがって，参照する側のデータ長が一致していれば問題ありませんが，参照側が短いときはエラー(警告)に，長すぎるときは先頭に＄00(ヌル・コード)が補われてアセンブルされます．また，dc.bで参照する場合，定義が複数の字数からなる文字列ならば，1バイトとして扱われるためエラー(警告)となります．

レジスタ・リスト(命令で指定するレジスタ名を並べたもの)を定義したいときは，**reg**擬似命令を使います．レジスタ名は通常データ・レジスタ，アドレス・レジスタの順に置き，同種のレジスタの番号が連続するときは“-”で中間のレジスタ名を省略することができます．そして，不連続な部分の区切りは“／”を使います．たとえば，

```
regs    reg         dl／a0-a3
        ⋮
        movem.l     regs,-(sp)
```

のように使います．このような複数レジスタを参照するのは，movem命令だけです．

1-4 データの定義

❖関連コマンド　dc, ds, even

ここでは，データ・セクションなどのデータ関係やセクション内における個々のデータの定義の仕方について述べます．

データを定義する擬似命令では，バイト(.b)，ワード(.w)，ロング・ワード(.l)のいずれのデータ長を採用するかを明確に記述しなければなりません．

●初期値をもつデータ(dc)

プログラムで参照する定数または初期値をもつ変数は，**dc**擬似命令で定義します．データ値は数値または文字値をとることができ，数値は10進のほかに16進値が指定できます．このとき，ラベルなどを参照する式によって間接的に数値を代入させる方法も用意されています．

各タイプの定義例を示すと次のようになります．

```
dc.w    100     ······10進値100をワード長で
dc.b    ＄0d    ······16進値0dをバイト長で
dc.l    factor······factorで定義されている値をロング・ワード長で(内容値はfactorの定義によって
                      決まる)
```

それぞれ確保します．

なお，文字列の場合は複数バイトになりますが，

```
dc.b   'string'
```

のようにバイト長を指定します．

dc 擬似命令では，同型のデータが続く場合，

```
dc.b   'Read   Error', $0d, $0a, 0
```

のようにカンマ(,)で区切って定数値を並べる方法で，1つの擬似命令にまとめることができます．

●初期値をもつデータ・ブロック(dcb)

同じ値をもつ同型データが連続する場合は，**dcb** 擬似命令が使えます．この命令の記述は，

```
dcb.b   5, '␣'   (␣はスペース)
```

のように，**繰り返し回数**，**データ値**の順序で並べます．この場合，5バイトのスペースがとられます．

●非初期化データ(ds)

初期値をもたない変数や一時的に使用される作業領域を確保するときは，その領域の大きさのみを定義すれば充分です．このようなときに使用するのが **ds** 擬似命令で，

```
ds.b 256
```

のように記述します．このケースでは256バイトの領域が確保されます．

●偶数アドレスへの配置(even)

68000では，奇数アドレスからワード，ロング・ワードのデータが始まることは許していません．しかし，一方でバイト長のデータを定義すると，後続のワード，ロング・ワードのデータが奇数バイトから始まることがあるため，もしそうならば現在のロケーションを1だけ加えて偶数アドレスに補正する even 擬似命令を使用します．

1-5 リンケージ・シンボルの定義と参照

❖関連コマンド　　**globl, xdef, xret**

プログラム・モジュール間のシンボルの参照は，リンカの段階で行なわれます．アセンブル時には，そのモジュール内で参照できなかったシンボルを外部参照とするために，/U スイッチが用意されています．また/D スイッチで全シンボルの外部からの参照を可能とする機能があるため，これらを利用する限りは他の特別な定義を必要としません．しかし，あまりこのような方法に頼りすぎると，プログラムの記述上の誤りによって予期していなかったシンボルを参照し，しかもそれがリスト上でのエラーのチェックにつながらないため，デバッグに手間どるといった弊害が発生します．とくに/D スイッチの使用は，外部から参照できるシンボル数を不必要に増加させるので慎重にしなければなりません．

こういったトラブルを避けるためには，リンカの扱うべき外部名を明確にし，限定するのが適当です．このための擬似命令には，

```
globl  <ラベル> { , <ラベル>}
```

があります．ここで指定されたラベルは，そのモジュール内に存在していれば外部から参照が可能で，な

いときは外部を参照します．

ただしこの方法でも，定義し忘れたりまたは，誤定義によって外部を参照するかされるかという違いが生ずるので，もっと厳格にしたいときのために次のような区別を行なう擬似命令が用意されています．

すなわち定義側における外部名の宣言には，

```
xdef 〈ラベル〉{ , 〈ラベル〉}
```

を使います．また，参照側での外部名の宣言には，

```
xref 〈ラベル〉{ , 〈ラベル〉}
```

を用います．

1-6 アドレッシング・モード

機械語命令には，どのような処理をどこに対して行なうかという情報が直接的，間接的に含まれています．このうち，どこに対して行なうかという情報の与え方にはいろいろな方法があり，その個々の形態を**アドレッシング・モード**と呼んでいます．

アドレッシング・モードは大別してレジスタ直接，メモリ・アドレス，特殊アドレスの３種類に分類されます．

このうち**レジスタ直接**は，処理の対象がレジスタになるもので，どのレジスタに特定するかという情報を与えます．レジスタの種類によって，**データ・レジスタ直接**，**アドレス・レジスタ直接**に分けられます．

メモリ・アドレスに分類されるタイプでは，メモリを参照するのにアドレス・レジスタを使い，間接的に対象を指示します．この形式では，基本的に対象アドレスは実行時まで確定しません．最も簡単な**アドレス・レジスタ間接**では，

```
(a0)
```

のように記述し，実行時にはA0レジスタに収容されている値がアドレス値として採用されます．さらに，この値に固定値を加えたいときは，**ディスプレースメント付きアドレス・レジスタ間接**を使い，

```
5(a0)
```

のように記述できます．また，変数を加えたいときは，変数を他の汎用レジスタ(この場合「インデックス・レジスタ」という)で与え，

```
5(a0, d0)
```

のように記述します．このような形式は，**インデックス付きアドレス・レジスタ間接**と呼ばれます．このほかに，実行後にアドレス・レジスタ値にデータ長を自動加算する**ポストインクリメント付きアドレス・レジスタ間接**，実行前に自動減算する**プリデクリメント付きアドレス・レジスタ間接**があり，それぞれ

・(a0)＋　……ポストインクリメント
・－(a0)　……プリデクリメント

のように記述されます．

その他の形式はすべて**特殊アドレス・モード**に分類されます．このうち**絶対アドレス**はアドレスとして固定値を指定するもので，ロング・ワードで指定するロング型と，ワードで指定するショート型の２種類があります．後者は結果的に符号拡張されて実行されるので，有効範囲は０～7FFF，FF8000～FFFFFFとなります．

PC相対形式は，実行時点でのPC(プログラム・カウンタ)値にディスプレースメント，インデックス値

を加算して対象アドレスを求めるもので，このタイプも実行時までアドレスが確定しません．ディスプレースメントのみのときは16ビット符号付き2進値，インデックス付きのときは8ビット符号付き2進値の定数が加算できます．

イミディエイト形式は，アドレスの代わりに固定値のデータをもち，その値が命令実行の際のソースとして採用されるものです．定数はバイト，ワード，ロング・ワードのいずれかの形態をとることができますが，バイト・データについても機械語の段階でワード・データと同じ命令長となり，上位バイトにゼロが充てんされる点に注意が必要です．より効率的にするには，バイト・データのみを与える**クイック・イミディエイト**形式が使える命令の場合，こちらを利用するのが適当です．この形式では，命令コードの中に8ビットで定数を押し込むようにアセンブルされ，実行時間もその分だけ短縮されています．

表1.3にアドレッシング・モードの一覧，図1.5に個々のモードの実効アドレス計算関連を示します．

●図1．5　各アドレッシング・モードの実効アドレス計算①

(a)　データ・レジスタ直接

実行アドレス計算　EA—Dn
アセンブラ形式　Dn
命令コード内MODE　000
命令コード内レジスタ番号　n

31　0
データ・レジスタ　Dn　実行対象

(b)　アドレス・レジスタ直接

実行アドレス計算　EA—An
アセンブラ形式　An
命令コード内MODE　001
命令コード内レジスタ番号　n

31　0
アドレス・レジスタ　An　実行対象

(c)　アドレス・レジスタ間接

実行アドレス計算　EA—(An)
アセンブラ形式　(An)
命令コード内MODE　010
命令コード内レジスタ番号　n

31　0
アドレス・レジスタ　メモリ・アドレス値
メモリ・アドレス　実行対象

(d)　ディスプレースメント付きアドレス・レジスタ間接

実行アドレス計算　EA＝(An)＋d_{16}
アセンブラ形式　d_{16}(An)，or (d_{16}，An)
命令コード内MODE　101
命令コード内レジスタ番号　n

31　0
アドレス・レジスタ An　メモリ・アドレス値
31　0
ディスプレースメント　符号拡張ずみ整数値　＋
メモリ・アドレス　実行対象

(e)　インデックス付きアドレス・レジスタ間接

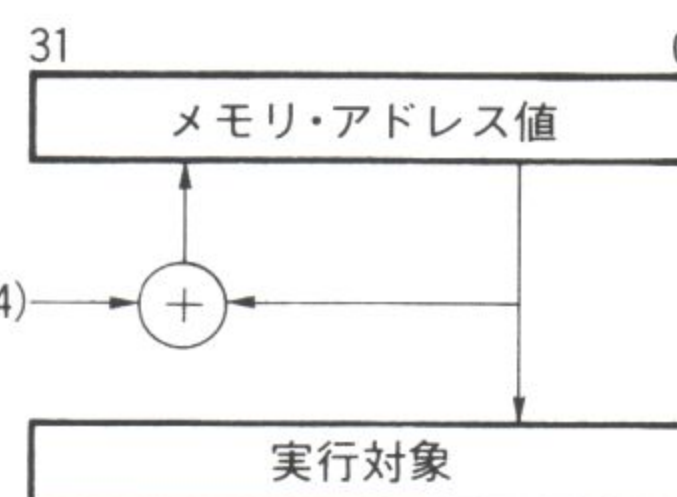
実行アドレス計算　EA＝(An)＋Xn＋d_8
アセンブラ形式　d_8(An,Xn.W) or (d_8,An,Xn.W)
d_8(An,Xn.L) or (d_8,An,Xn.L)
命令コード内MODE　110
命令コード内レジスタ番号　n

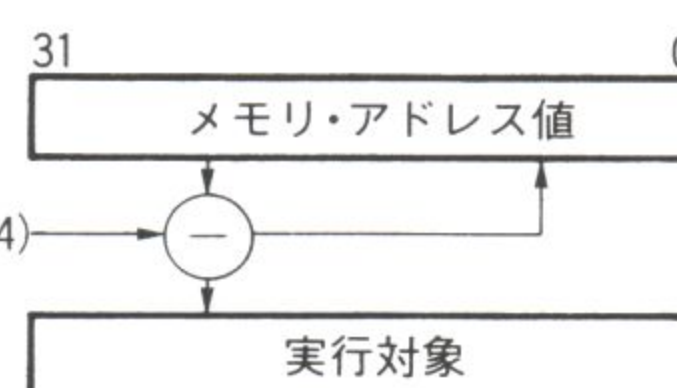
31　0
アドレス・レジスタ　メモリ・アドレス値
31　0
ディスプレースメント　符号拡張ずみ整数値　＋
31　0
インデックス・レジスタ　符号拡張ずみ整数値　＋
メモリ・アドレス　実行対象

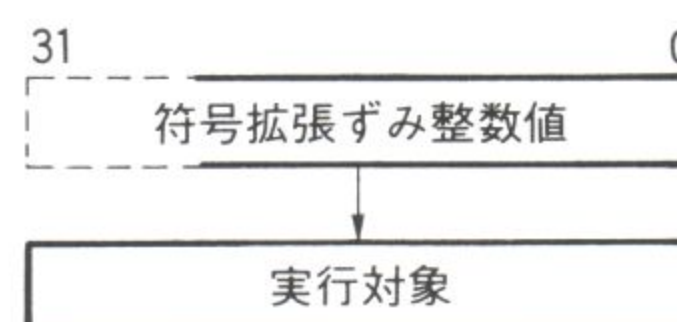
●表1．3　68000のアドレッシング・モード一覧

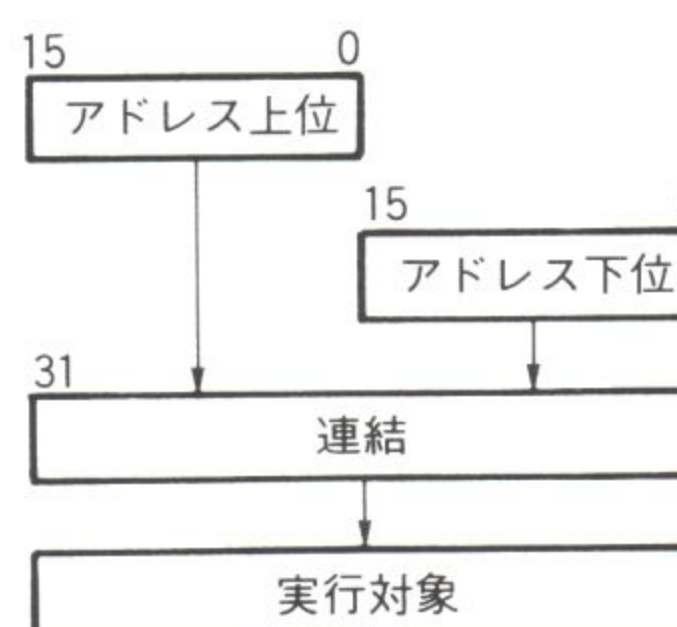
Addressing Mode	Mode	Register
データ・レジスタ直接	000	Register Number
アドレス・レジスタ直接	001	Register Number
アドレス・レジスタ間接	010	Register Number
ポスト・インクリメント付きアドレス・レジスタ間接	011	Register Number
プリデクリメント付きアドレス・レジスタ間接	100	Register Number
ディスプレースメント付きアドレス・レジスタ間接	101	Register Number
インデックス付きアドレス・レジスタ間接	110	Register Number
絶対アドレス(ショート)	111	000
絶対アドレス(ロング)	111	001
ディスプレースメント付きPC相対	111	010
インデックス付きPC相対	111	011
イミディエイト	111	100

(f)　ポスト・インクリメント付きアドレス・レジスタ間接

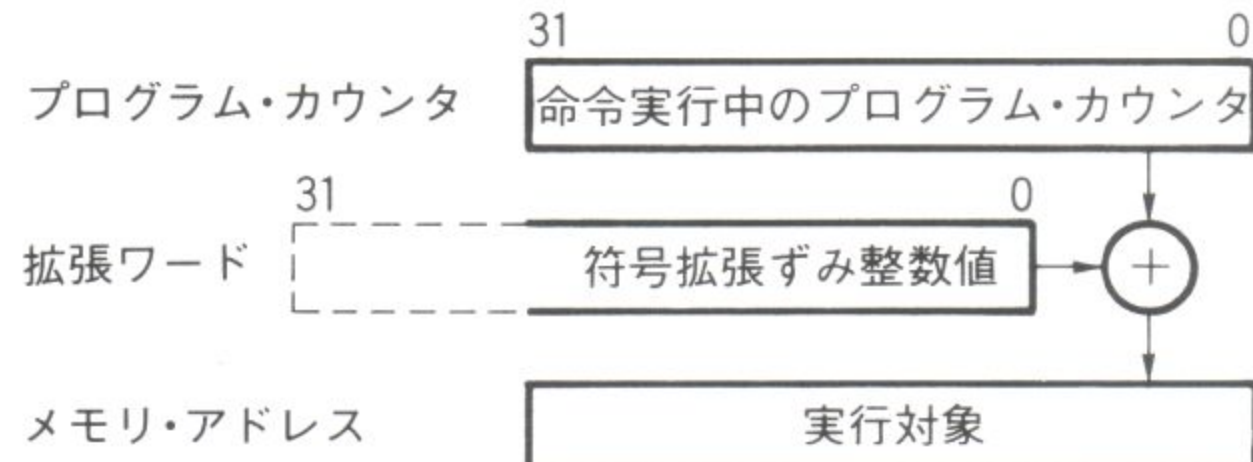
実行アドレス計算　An←An＋N
アセンブラ形式　(An)＋
命令コード内MODE　011
命令コード内レジスタ番号 n

(g)　プリデクリメント付きアドレス・レジスタ間接

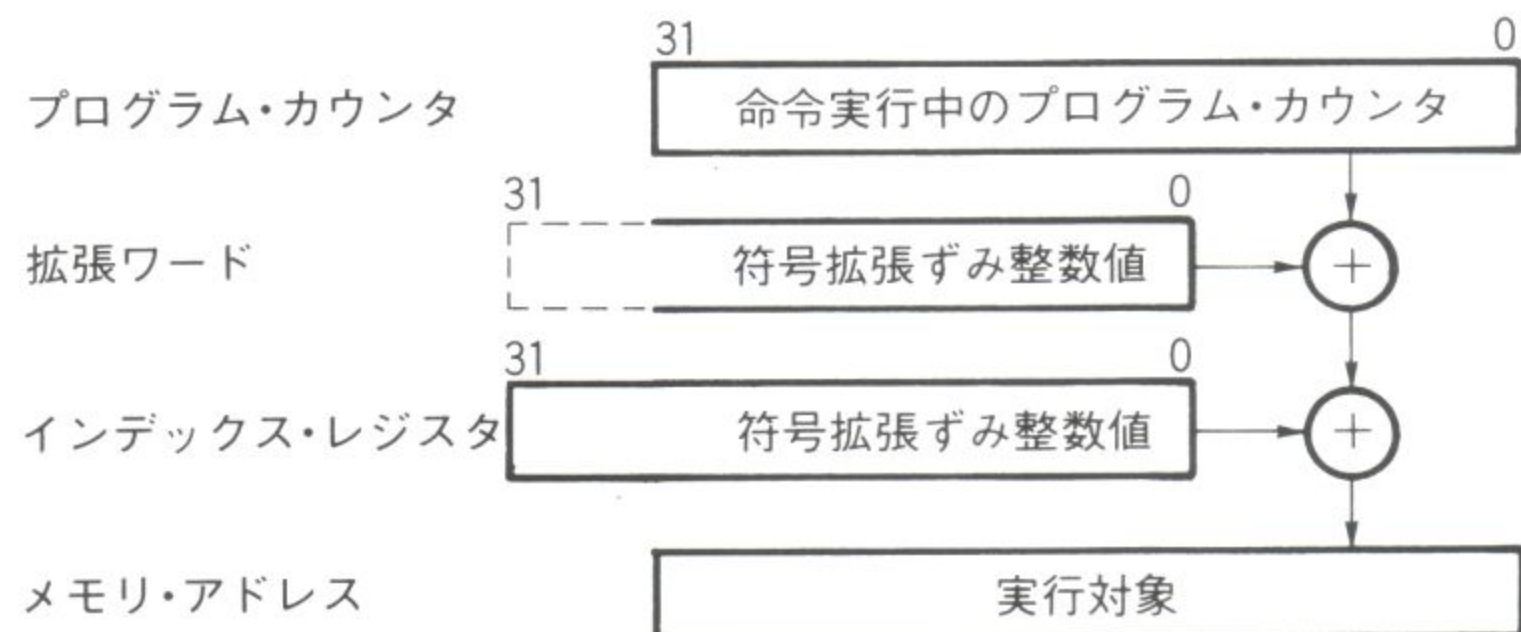
実行アドレス計算　An←An－N
EA←(An)
アセンブラ形式　－(An)
命令コード内MODE　100
命令コード内レジスタ番号 n

(h)　絶対アドレス(ショート型)

実行アドレス計算　EA GIVEN
アセンブラ形式　xx.W or (xxx.W)
命令コード内MODE　111
命令コード内レジスタ番号 000

(i)　絶対アドレス(ロング型)

実行アドレス計算　EA GIVEN
アセンブラ形式　xxx.L or (xxx.L)
命令コード内MODE　111
命令コード内レジスタ番号 001

●図1．5　各アドレッシング・モードの実効アドレス計算②

(j)　ディスプレースメント付きPC相対

実行アドレス計算　EA＝(PC)＋d_{16}
アセンブラ形式　d_{16}(PC)　or　(d_{16}，PC)
命令コード内MODE　111
命令コード内レジスタ番号　010

(k)　インデックス付きPC相対

実行アドレス計算　EA＝(PC)＋(Xn)＋d
アセンブラ形式　d_8(PC，Xn.W)　or　(d_8，PC，Xn.W)
d_8(PC，Xn.L)　or　(d_8，PC，Xn.L)
命令コード内MODE　111
命令コード内レジスタ番号　011

(l)　イミディエイト

実行アドレス計算　OPERAND GIVEN
アセンブラ形式　＃xxxx or＃<data>
命令コード内MODE　111
命令コード内レジスタ番号　100

1-7 プログラム・セクション

❖関連コマンド　**text, data, bss, stack**

プログラムには，機械語命令のほかに，データを収容する領域が必要です．X68000のアセンブラでは，データの種類によってさらに3つのタイプの領域に分割し，合計4種類の領域(セクション)を設定しています．各セクションのあらましは次のようなものです．

●テキスト・セクション

.text擬似命令に続く領域で，機械語命令群を収容します．

●データ・セクション

データのうち,主として初期値をもつものを収容する領域です..data擬似命令がこの領域の先頭を表わします．

●ブロック・トレース・セクション

データのうち，初期値をもたないもので，かつ実行開始時点で必要なサイズが確保ずみであることを前提とする領域です．いわば「静的」に割り付けする作業領域で，プログラム間で共通に参照する領域は，データ・セクションか，ブロック・トレース・セクションのいずれかでなければなりません．この領域の先頭は,.bss擬似命令で定義されます．

●スタック・セクション

データの中でも，サブルーチンで一時的に使用するなど，プログラム実行開始時点でとくに確保されていなくてもよい部分がこれにあたります．この領域は，プログラムの命令実行中に「動的」に取得，解放されるものであるため，複数のプログラムが共有する領域としては適当ではありません．また，領域の取得，解放手続きは，テキスト・セクション中の命令によってプログラマが意識的に行なうことになります．この領域の先頭は,.stack擬似命令で定義されます．

X68000のアセンブラは，以上の各セクションについて，とりあえず0番地から順にアドレスを割り付けます．こうしておいて，リンカの段階で同じセクションのものを寄せ集め，最終的なアドレスを決定します．

1-8 前方参照と外部参照

アセンブラは，ソース・プログラムの命令行を順序よく読み取り，機械語に変換していきます．この処理(パス)は2段階に分けて行なわれ，1回目はラベルに対するアドレスなど具体的な値の決定，2回目は機械語の生成がなされます．

ラベルから具体的な値に変換するパスでは，その途中で参照すべきラベルが未処理，すなわちこれから処理する部分に定義されているというケースがあり，これを**前方参照**といいます．ステップとしては後方にあるのですが，処理の進行方向という観点からは前方にあるのでこのように呼ばれるのです．

前方参照が生じた場合，その時点ではすぐに最終的な定義を参照できないため，具体的な値の決定は保留されます．命令(擬似命令を含む)によっては，保留することでアドレス値などの決定ができなくなる場合があるので，その危険があるケースでは前方参照を禁止しています．支障のないケースについては，未

定義の場合を除き，第１ステップ完了と同時に参照値がすべて確定するので禁止する理由がありません．ただし，前方参照した先でさらに前方参照すると，アセンブラが最終的な参照値を確定するのが難しくなります．したがって，許される場合でも１回だけに限定される点に注意しなければなりません．

外部参照は，他のプログラム・モジュールの定義を参照することをいい，現在処理中のモジュールの中にその定義が存在しないことを意味します．たとえば，他のモジュールの中に存在するサブルーチンを実行する場合，そのサブルーチン名は外部参照になります．

外部参照については，アセンブルする段階では具体的なアドレス値が得られないのですが，サブルーチンにジャンプする命令でそのことを考慮しさえすれば，アセンブルそのものについては支障をきたしません．しかし，参照値によってはアセンブル結果に影響する（たとえばアドレスの割り付けが変わるなど）場合は許されません．

1-9　アセンブル・リストの部分的出力停止

❖関連コマンド　　.nlist, .list

アセンブル・リストは，プログラムの最終ドキュメントとして，デバッグなどに不可欠なものです．しかし，情報という観点から見た場合，量ばかり多くてはかえって作業の妨げになるため，すでにわかりきっている部分のリスティングは省略するのが一番よいでしょう．

たとえば，include擬似命令で挿入されるファイルの内容については，あくまで承知の上で引用しているのですから，リスティングの段階では不必要です．

このようなとき，リスティングを停止するのが，

```
.nlist
```

擬似命令です．アセンブラは，この命令を検出した後，その行からのリスティングを省略します．

一方，停止中のリスティングを再開するには，

```
.list
```

擬似命令を使います．この擬似命令により，リスティングはその行から再開されます．

これらの擬似命令を使ってincludeされるファイルの内容のリスティングをバイパスするには，引用する側でリスティングの制御をするよりも，引用される側でコントロールするのが合理的です．なぜなら，一度引用される側でリスティング制御の記述をするだけで，引用の都度バイパスのための手続きをしなくてもすむからです．また，引用する側でバイパスさせるには，nlist擬似命令を使ったあとでinclude擬似命令を使うことになります．そうすると，include擬似命令の内容がリスティングされないため，リストを見てもどのファイルが挿入されたのかがわかりません．

なお，list擬似命令が使われても，アセンブラ立ち上げ時に／p（リスト・ファイルの指定）スイッチを使わなければリスティングは行なわれません．

1-10　条件付きアセンブル

❖関連コマンド　ifxx, elseif, endif

プログラムの一部について，条件によってアセンブルをバイパスしたり，異なった内容でアセンブルを行なうことができます．

表1.4はそのための擬似命令で，一般に if××～(else～)endif の形式で用いられます．elseif は，その前の if×× が成立しないとき，残りの条件を検査するのに使われます．それでも条件が成立しないものには，さらに else 以下の内容を適用できます．

具体的には，

```
ifdef    DEBUG
      (a)
else
      (b)
endif
```

では，DEBUG が定義されているとき a がアセンブルされ，そうでないとき b がアセンブルされます．b がないときは else も省略できます．

●表1．4　条件付きアセンブルのための擬似命令一覧

形　式	意　味
if(ifne)　〈式〉	条件が真のときアセンブル実行
iff(ifeq)　〈式〉	条件が偽のときアセンブル実行
ifdef　〈シンボル〉	シンボルが定義されているときアセンブル実行
ifndef　〈シンボル〉	シンボルが定義されていないときアセンブル実行
else	反対の条件が存在するときアセンブル実行
elseif　〈式〉	反対の条件が存在し，かつ特定の条件を満たすときアセンブル実行
endif	条件付きアセンブルの終了

第2章

68000命令セットの概要

2-1 データ値のコピーを作る命令

❖関連コマンド　　**move, movea, moveq, movep, exg, swap**

レジスタまたはメモリ(周辺装置を含む)の相互間で，データのコピーを行なう処理は，きわめて高い頻度で行なわれています．この処理によってデータが移動するので，そのとき使われる命令は「移送命令」と呼ばれています．

68000では，**移送命令**として**move**が使われます．moveの一般的な形式は，

```
move | .b |
     | .w | <ソース ea>, <デスティネーション ea>
     | .l |
```

のように，移送元を左に，移送先を右に書きます．他の命令でもそうですが，68000ではデータの流れが，通常左側のオペランドから右側のオペランドに向かうように書くことになっています．

ソースとデスティネーションの組み合わせでは，メモリ同士の指定をすることによって，メモリからメモリにレジスタを介さず直接コピーができる点が非常に強力です．レジスタを指定するときは，データ・レジスタ，アドレス・レジスタのほかに**CCR**(コンディション・コード・レジスタ)も指定でき，スーパーバイザ・モードでは**SR**(ステータス・レジスタ)や**USP**(ユーザ・スタック・ポインタ)も扱えます．

データ長は，普通move命令に続く指定でバイト(.b)，ワード(.w)，ロング・ワード(.l)の3通りが選べますが，アドレス・レジスタとのやり取りに際しては，バイト・サイズが指定できません．そして，アドレス・レジスタに転送する場合は，ソースがワード・サイズであっても，符号拡張されたロング・ワード・サイズで書き込みが行なわれます．このため，アセンブラでは特別に**movea**命令を用意し，その場合

68000はたくさんの命令をもっています．筆者の持論としては，それらの命令を，個々の内容で覚えるよりも，グループ単位で見た方が理解が早いと考えています．そこで本章では，性質の類似している命令をとりまとめ，グループごとに紹介することにします．

説明の中に登場する〈ea〉は実効アドレスのことで，通常メモリを参照するすべてのアドレッシングを含みます．それ以外に，データ・レジスタ，アドレス・レジスタ直接が使えるときは注釈を付けました．

また，各命令実行後のレジスタやフラグの変化は重要なことが多いので，命令の動作を想定しながら設定値を予測する訓練をしておくと，効率のよいプログラムが作れるようになります．

なお，命令コードのビット構成などについては，デバッグ時に逆アセンブラを使えば参照する必要がないので省略しました．

movea | .w
.l | 〈ea〉, An

のように書きます．

ソースとして定数を用いるときは，イミディエイト・アドレッシングによって任意のサイズが選択できます．しかし，バイト・サイズならば **moveq** 命令を使うほうが機械語命令の長さが短く，その分だけ高速に実行できます．この命令では，1バイトのデータが符号拡張されてロング・ワード・サイズの形で移送される点に注意が必要です．また，デスティネーションはDnしか指定できません．

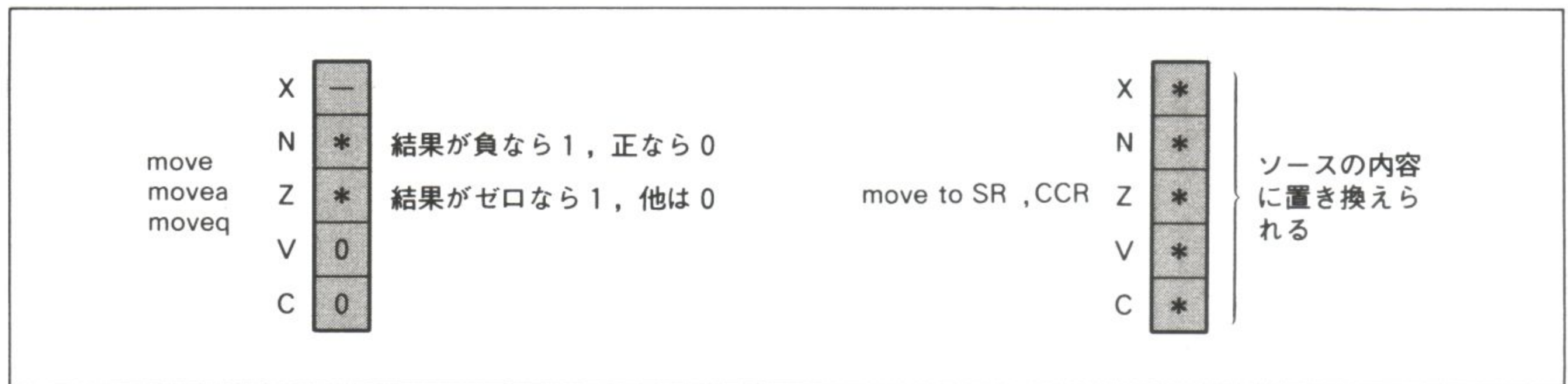
変わったところでは，**movep** 命令があります．この命令はソースで指定されたデータ・レジスタのデータをバイト単位に分割し，デスティネーション(メモリ)のアドレス上で1バイト置きに移送します．または，反対に1バイト置きに並んでいるソースのデータ(メモリ)を読み出し，指定されたデスティネーション(データ・レジスタ)に連結して並べます．主として周辺装置とのデータのやり取りに使われ，メモリのアドレスはディスプレースメント付きアドレス・レジスタ間接によって指定されます．

また，複数のレジスタのメモリへの退避や復帰に使われる **movem** 命令については，「スタック操作命令」のところで述べます．

データのコピーの特殊なケースとして，**交換**(exchange)があります．データ交換命令としては，レジスタ同士に限定された

exg | Dn
An | , | Dn
An |

●図2．1　移送命令実行後のフラグの変化（move from SR. CCR と movep, exg, swap では変化しない）

命令	フラグ	値	説明
move movea moveq	X	—	
	N	*	結果が負なら1，正なら0
	Z	*	結果がゼロなら1，他は0
	V	0	
	C	0	
move to SR ,CCR	X	*	ソースの内容に置き換えられる
	N	*	
	Z	*	
	V	*	
	C	*	

が用意されています．

また，同一データ・レジスタ内で上位ワードと下位ワードの内容を交換する

```
swap    Dn
```

命令もあります．

以上の各命令実行後のフラグの変化は図2.1のとおりです．

2-2 加減算命令

❖関連コマンド　**add, adda, addq, addx, sub, suba subq, subx, abcd, sbcd, lea, pea**

move 命令は，68000の命令の中でもとくに広範囲なアドレッシング・モードの組み合わせに対応できるように設計されています．しかし，他の命令では，命令の性格によりカバーする範囲が狭くなっているので注意が必要です．

加減算については，基本的なものとして加算命令(**add**)と減算命令(**sub**)があり，これらはデータ・レジスタと実効アドレスが示す対象との間で演算を行ないます．アドレス・レジスタを対象とするものは，**adda**, **suba** があり，これらは実効アドレスが示す対象をソースとして，アドレス・レジスタへの足し込み，引き去りを行ないます．言い換えると add と sub は双方向の加減算が可能ですが，adda と suba は一方向のみの演算の流れとなっています．

演算時には定数を加算あるいは減算することも少なくありませんが，この場合3ビットの符号なし定数が使える **addq**, **subq**(**クイック**加算，減算)と，8，16，32ビットの符号付き定数が使える **addi**, **subi**(**イミディエイト**加算，減算)があります．クイック加減算は指定数値範囲が狭い欠点がありますが，命令が1ワードのみと短く，実行速度が速いのでこのように呼ばれています．イミディエイト加算は命令コードのほかに拡張ワード(2バイト)またはロング・ワード(4バイト)が必要で，その分だけ遅くなりますが，データ領域を参照して取り出すよりは高速です．クイック加減算では，実効アドレスとしてアドレスおよびデータ・レジスタ直接が，イミディエイト加減算ではデータ・レジスタ直接が指定できます．

演算時につきものの桁上がり，桁下がりについては，CCR の C(キャリー)フラグと X(拡張)フラグにセットされます．C フラグは move 命令などによっても変化しますが，X フラグは演算命令でしか変化しないので，注意してプログラミングすれば次の上位桁の演算を継続するときまで有効にすることができます．X フラグをキャリー(引き算のときはボロー)とみなして加減算を行なう命令には **addx**, **subx** がかります．

以上の加減算命令では2進で処理されますが，2進化10進で行ないたければ **abcd**, **sbcd**(BCD 加算，減算)命令を使うこともできます．ただし BCD 加減算については，データ・レジスタ同士，ポストデクリメント付きアドレス・レジスタ間接によるメモリ同士の演算形態しか許されていません．

加減算の特殊な形態の1つに，実効アドレスの計算があります．レジスタ直接などを除き，インデック

●図2．2　加減算命令実行後のフラグの変化

add, sub, adda, suba, addq, subq, addi, subi, addx, subx

フラグ	値	説明
X	*	Cフラグと同じ
N	*	結果が負なら1，正なら0
Z	*	結果がゼロなら1，他は0
V	*	オーバーフローしたら1，他は0
C	*	桁上がり(下がり)が生じたら1，他は0

abcd, sbcd

フラグ	値	説明
X	*	Cフラグと同じ
N	U	無意味
Z	*	結果がゼロなら1，他は0
V	U	無意味
C	*	桁上がり(下がり)が生じたら1，他は0

スト・アドレッシングなどのメモリのアドレスは，実行時に計算された結果の値が用いられます．その値をアドレス・レジスタに貰い受ける命令として，**lea**(ロード・エフェクティブ・アドレス)，スタック(メモリ)に収容する命令として**pea**(プッシュ・エフェクティブ・アドレス)が用意されています．これらはアドレス値の計算に用いられたり，インデックス付きアドレス・レジスタ間接形式で定数，アドレス・レジスタ，インデックスに指定したレジスタの三者を一度に加算するなどの特殊用途に利用されています．

以上の各命令実行後のフラグの変化は図2.2のとおりです．

2-3　乗除算命令

❖関連コマンド　mulu, muls, divu, divs

68000の乗算，除算命令は，符号付きと符号なし2進数に対応できるように，それぞれ2種類の命令をもっています．乗算では**mulu**(符号なし)と**muls**(符号付き)，除算では**divu**(符号なし)と**divs**(符号付き)がそれらに該当します．

演算はデータ・レジスタをデスティネーションとして行なわれるため，乗算では結果が32ビットで得られるようになっています．このため，16ビット×16ビットの計算を行ないます．除算についても，32ビット÷16ビットの結果が16ビットとなります．これは，デスティネーションとなるデータ・レジスタは，演算前の数値の「容れ物」であると同時に，結果を収容するのにも使われるため，上限の32ビットで「頭打ち」になるからです．

これらの命令記述は，デスティネーションがデータ・レジスタに限定されていることから，

```
mulu
muls
divu     <ea>, Dn
divs
```

の形式しか存在しません．

以上の各命令実行後のフラグの変化は図2.3のとおりです．

●図2．3　mul, div関係命令実行後のフラグの変化

mulu, nuls, divu, divs

フラグ	値	説明
X	—	
N	*	結果が負なら1，正なら0
Z	*	結果がゼロなら1，他は0
V	*	オーバーフローしたら1，他は0
C	0	

(注) 除算においてゼロで割ったときは，N，Zフラグは未定義(無意味)

2-4 補助演算命令

❖関連コマンド　　**clr, ext, neg, negx, nbcd**

ここでは，２進あるいは２進化10進演算時に補助的に使われる命令についてとりあげます．これらの命令は，演算前の準備段階で，あるいは演算処理過程で必要に応じて使われるものです．

演算前に行なわれるゼロ・クリアのためには，

clr ｜ .b / .w / .l ｜ 〈ea〉（ea はデータ・レジスタ直接を含む）

命令がよく使われます．

また，データ・レジスタ値の**有効ビット幅**を倍に**拡張**する

ext ｜ .w / .l ｜ Dn

も演算の過程で補助的に使われます．この命令は，上位となるビットに，下位データの最上位ビットを符号として転送するもので，命令のデータ長は結果側のものを指定します．

補数化命令としては，**２進補数**用の

｜ neg / negx ｜ .b / .w / .l ｜ 〈ea〉（ea はデータ・レジスタ直接を含む）

と，**２進化10進補数**用の

nbcd 〈ea〉　（ea はデータ・レジスタ直接を含む）

があります．**negx** は，上位桁に拡張するときに，下位からのキャリーを **X フラグ**から受け取る点が neg と異なります．**nbcd** も Xフラグを参照するため，拡張を意図していることがわかります．したがって，初回（拡張する前の演算）は前もって Xフラグを 0（10の補数の場合）にしておく必要があります．もし X＝1で

●図2．4　補助演算命令実行後のフラグの変化

clr

フラグ	値
X	—
N	0
Z	1
V	0
C	0

ext

フラグ	値	説明
X	—	
N	＊	結果が負なら１，正なら０
Z	＊	結果がゼロなら１，他は０
V	0	
C	0	

neg

フラグ	値	説明
X	＊	Cフラグと同じ
N	＊	結果が負なら１，正なら０
Z	＊	結果がゼロなら１，他は０
V	＊	オーバーフローしたら１，他は０
C	＊	桁下がりが生じたら１，他は１

nbcd

フラグ	値	説明
X	＊	Cフラグと同じ
N	U	無意味
Z	＊	結果がゼロなら１，他は０
V	U	無意味
C	＊	10進で桁下がりが生じたら１，他は０

演算を開始すると9の補数となります．
以上の各命令実行後のフラグの変化は図2.4のとおりです．

2-5 ビット処理命令

❖関連コマンド　**asl, asr, lsl, lsr, rol ror, roxl, roxr, not, and or, eor, andi, ori, eori**

ビット処理のための命令としては，シフト，ローテートといったビット間の移動を行なわせるものと，論理演算のための命令ブロックをあげることができます．

シフトには算術的とロジカル的の2種類があります．前者は**asl**(**算術的左シフト**：1回のシフトで数値が2倍になる)と**asr**(**算術的右シフト**：1回のシフトで数値が半減する)のいずれも，シフトすることによって数学的な意味を伴うので，こう呼ばれます．後者の**lsl**(**ロジカル左シフト**)と**lsr**(**ロジカル右シフト**)は，シフトを開始するビットには0が詰められる形でシフトを行なうもので，lslとaslは同じです．

ローテートにも2種類あって，このうち**rol**(**左ローテート**)と**ror**(**右ローテート**)のグループは最上位ビットと最下位ビットがつながった形でデータ・ビットの回転を行ないます．それに対して，**拡張ローテート**は，シフトの拡張のため用意された命令のグループで，最上位ビットと最下位ビットの間に**Xフラグ**が入った形でリングを形成します．このため前段からのビットを，Xフラグを経由して後段に伝えることができます．このグループに含まれる命令のニーモニックには，**roxl**(**左拡張ローテート**)，**roxr**(**右拡張ローテート**)があります．

これらの命令には，1ビットだけシフト(またはローテート)を行なう，

```
| asl  |  | .b |
|  :   |  | .w |    <ea>
| roxr |  | .l |
```

の形式と，複数ビット(定数指定では1～8，レジスタ指定は0～63)シフト(またはローテート)できる次の形式があります．

```
| asl  |  | .b |  | # <定数> |
|  :   |  | .w |  |          |  , Dn
| roxr |  | .l |  | Dx       |
```

後者の形式では，シフトする回数が多いと時間がかかるため，デスティネーションとしてメモリを指定することができず，データ・レジスタのみに限定されています．

論理演算命令には，NOT，AND，OR，Ex-ORの演算を行なうものが用意されています．

このうちNOTは，

```
    | .b |
not | .w |   <ea>
    | .l |
```

形式だけですが，ANDとORについてはデータ・レジスタとメモリなどの実効アドレスにより指定される対象との間で，双方向に演算を行なえます．この点でEx-OR(排他的論理和)はソースがデータ・レジスタとなるパターンしか用意されていないので注意が必要です．使える形式を並べてみると，

```
| and |   | .b |
|     |   | .w |    <ea>, Dn
| or  |   | .l |
```

```
| and |   | .b |
| or  |   | .w |    Dn, <ea>
| eor |   | .l |
```

のパターンに整理できます．

また，ソースとして定数を指定したいときのために次のパターンも用意されています．

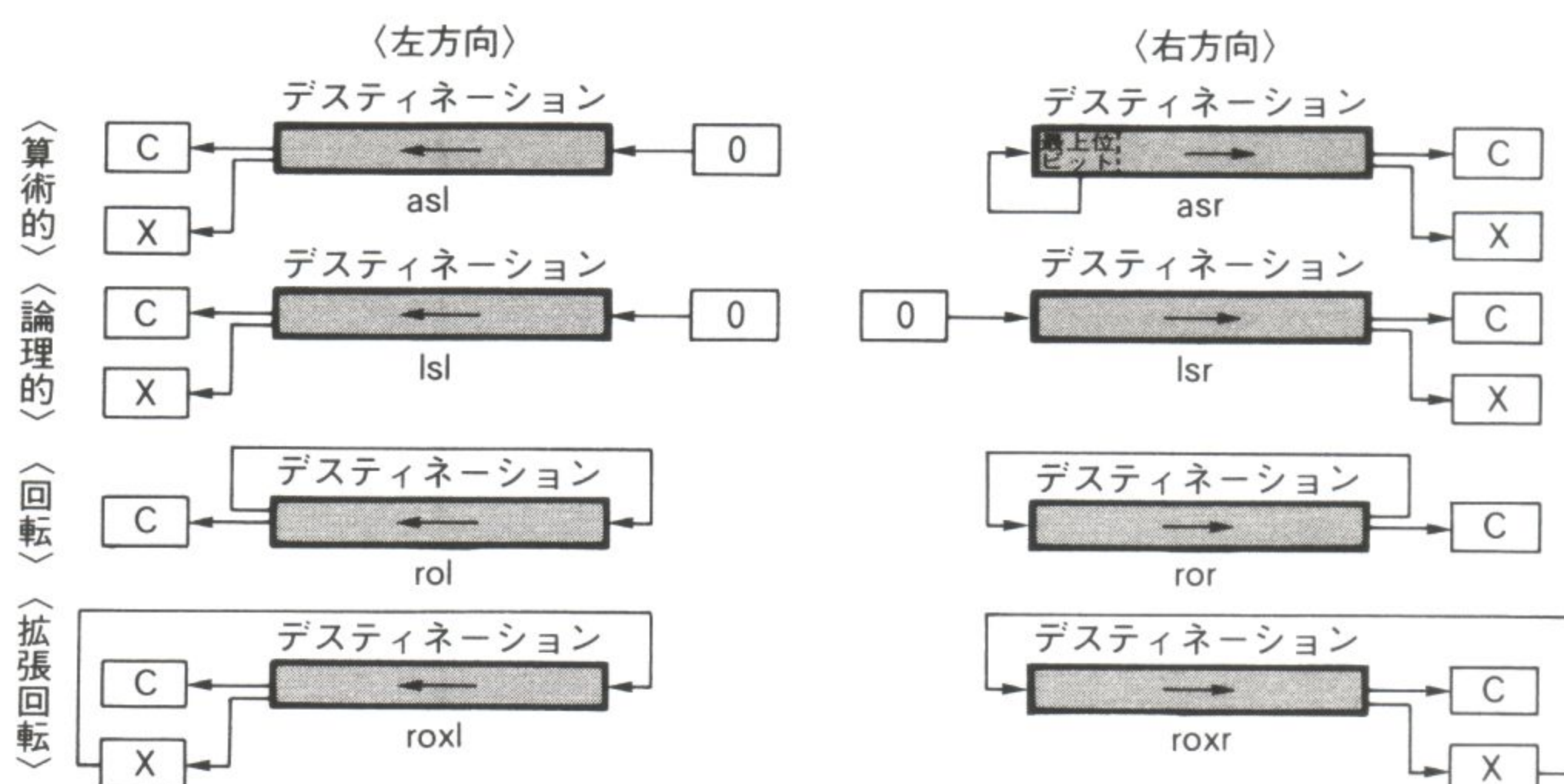
●図2．5　シフト，ローテートの動作と実行後のフラグの変化

〈左方向〉　〈右方向〉

〈算術的〉　デスティネーション　C　X　0　asl　デスティネーション　最上位ビット　C　X　asr

〈論理的〉　デスティネーション　C　X　0　lsl　0　デスティネーション　C　X　lsr

〈回転〉　デスティネーション　C　rol　デスティネーション　C　ror

〈拡張回転〉　デスティネーション　C　X　roxl　デスティネーション　C　X　roxr

命令	フラグ	値	説明
asl asr	X	*	命令によって押し出された値（シフト・カウント＝0のとき変化せず）
	N	*	結果の最上位ビットと同じ
	Z	*	結果がゼロなら1，他は0
	V	*	結果の符号が変化したら1，他は0
	C	*	命令によって押し出された値（シフト・カウント＝0のとき0）

命令	フラグ	値	説明
lsl lsr	X	*	命令によって押し出された値（シフト・カウント＝0のとき変化せず）
	N	*	結果の最上位ビットと同じ
	Z	*	結果がゼロなら1，他は0
	V	0	
	C	*	命令によって押し出された値（シフト・カウント＝0のとき0）

命令	フラグ	値	説明
rol ror roxl roxr	X	*	roxl，roxrは命令によって押し出された値（ローテート・カウント＝0およびrol，ror命令では変化せず）
	N	*	結果は最上位ビットと同じ
	Z	*	結果がゼロなら1，他は0
	V	0	
	C	*	命令によって押し出された値（ローテート・カウント＝0のとき0）

●図2．6　論理演算命令実行後のフラグの変化

not

	フラグ	
X	—	
N	＊	結果が負なら1，正なら0
Z	＊	結果がゼロなら1，他は0
V	0	
C	0	

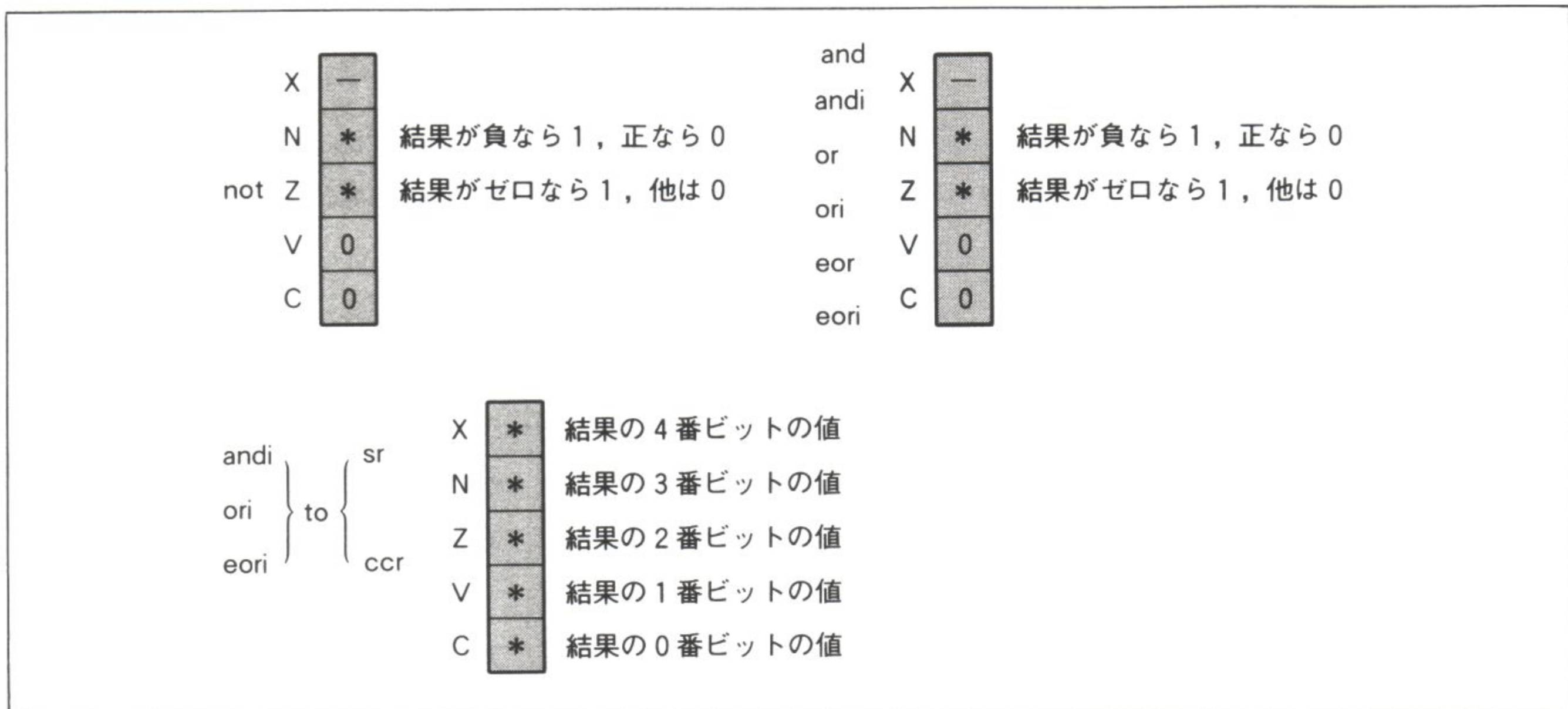
and / andi / or / ori / eor / eori

	フラグ	
X	—	
N	＊	結果が負なら1，正なら0
Z	＊	結果がゼロなら1，他は0
V	0	
C	0	

andi / ori / eori　to　sr / ccr

	フラグ	
X	＊	結果の4番ビットの値
N	＊	結果の3番ビットの値
Z	＊	結果の2番ビットの値
V	＊	結果の1番ビットの値
C	＊	結果の0番ビットの値

```
| andi |  | .b |
| ori  |  | .w |   #〈定数〉,〈ea〉
| eori |  | .l |
```

```
| andi |
| ori  |   [.w]   #〈定数〉, SR  （特権命令）
| eori |
```

```
| andi |
| ori  |   [.b]   #〈定数〉, CCR
| eori |
```

これらはイミディエイト・アドレッシングで，デスティネーションとしてSRやCCRを指定できます．ただし，SRを指定する命令は特権命令なので，ユーザ・モードでは使えないことに注意が必要です．

以上の命令のうち，シフト，ローテート命令実行後のフラグの変化を図2.5に，論理演算命令実行後のフラグの変化を図2.6に示します．

2-6　条件検査命令

❖関連コマンド　**cmp, cmpa, cmpi, tst, btst, bchg, bclr, bset, tas**

BASICやCでは，IF文で条件に応じた処理ができますが，機械語レベルでは比較した結果によって分岐するという，いわゆる条件分岐しかできません．条件を検査するための比較は，引き算を利用して行なわれ，デスティネーションの値からソースの値を差し引きます．しかし，結果はデスティネーションには格納されず，CCRのフラグが変化するだけです．条件検査命令には，次のパターンがあります．

```
cmp | .b |
    | .w |   〈ea〉, Dn
    | .l |
```

●図2．7　条件検査命令実行後のフラグの変化

cmp / cmpa / cmpi / cmpm

フラグ	値	説明
X	—	
N	*	結果が負なら1，正なら0
Z	*	結果がゼロなら1，他は0
V	*	オーバーフローしたら1，他は0
C	*	桁下がりが生じたら1，他は0

tst

フラグ	値	説明
X	—	
N	*	結果が負なら1，正なら0
Z	*	結果がゼロなら1，他は0
V	0	
C	0	

btst / bchg / bclr / bset

フラグ	値	説明
X	—	
N	—	
Z	*	結果がゼロなら1，他は0
V	—	
C	—	

tas

フラグ	値	説明
X	—	
N	*	結果が負なら1，正なら0
Z	*	結果がゼロなら1，他は0
V	0	
C	0	

```
cmpa  | .w |   〈ea〉, An
      | .l |

      | .b |
cmpi  | .w |   #〈定数〉, 〈ea〉
      | .l |

      | .b |
cmpm  | .w |   (An)+, (An)+
      | .l |

      | .b |
tst   | .w |   〈ea〉
      | .l |
```

上記の中でtstは，0からea(実効アドレス)が示す対象を引いて結果をCCRに反映させます．

似たような命令に，ビット・テストのグループがあります．その中でbtst(ビット・テスト)は結果をデスティネーションに格納しないAND命令で，以下bchg(ビット・テスト後反転)，bclr(ビット・テスト後ゼロ・クリア)，bset(ビット・テスト後1にセット)のバリエーションをもっています．このグループの書き方は，次のとおりです．

```
| btst |
| bchg |   #〈定数〉, 〈ea〉     (バイト・サイズ)
| bclr |
| bset |

| btst |
| bchg |   Dn, 〈ea〉          (ロング・ワイド・サイズ)
| bclr |
| bset |
```

特殊なものでは

```
tas <ea>
```

(test and set)があり，この命令は実効アドレスが示す対象と0とを比較してN, Zフラグをセットした後，最上位ビット(D_7)を1にセットするのでこの名があります．

以上の各命令実行後のフラグの変化を図2.7に示します．

2-7 実行順制御などの命令

❖関連コマンド　jmp, bra, bcc, dbcc, scc

プログラムの実行順を変更する命令には，強制的に変更する**無条件ジャンプ**と**ブランチ**，条件検査結果のフラグによって変更する**条件ブランチ**があります．

ここでいうジャンプとは，実効アドレスを値を転送することによってPC値を変更するもので，これに対しブランチはPC値に対し相対値を加算することによって変更します．したがって，ブランチ命令を使ったプログラムは，特別な工夫をしなくてもプログラムをポジション・インディペンデント(ロード位置が自由なこと)にできます．jmp(無条件シャンプ)，bra(無条件ブランチ)の書き方は次のとおりです．

```
jmp <ea>
bra [.s]    <ラベル>
```

ここでは飛び先のラベルを記述します．ブランチ先相対アドレスが8ビットで表現できるときは，**bra.s**とすることによって2バイト命令に縮められます．そうでないときは，相対アドレスは16ビットとなり，4バイト命令になります．

条件ブランチには，表2.1に示す種類があり，命令ニーモニックの“b”の次はCCRの状態を表わす記述が続くので，これらを総称して，あるいは代表してbccと書かれることがあります．これらの命令記述は，

```
bcc. [.s]    <ラベル>
```

形式で，詳細はbraと同じです．

ブランチ命令はループの底にも使われることが多く，このとき条件ブランチを使えば，cのdo～while的な動作ができます．ループはどちらかといえば所定のカウントに達するまで繰り返すといったパターンが多いので，機械語でも専用の命令が用意されています．

それは

```
dbcc [.s]    Dn, <ラベル>
```

で，ccはbccと同様に表2.1の内容が使われます．この命令は条件が成立しなかったときデータ・レジスタ(Dn)の値が1減らされ，－1にならなければ，ラベル指定された位置にブランチします．成立時または1減によってDnが－1になったときは，次の命令に移ります．最も多く使われるのは**dbra**で，このループはDnの初期値＋1回だけ繰り返しが行なわれます．

さて，条件による処理の中には，結果によって単に0か1かを転送するようなケースがあります．このようなときは，

```
scc <ea>
```

を使用すると，実効アドレスで指定される対象には，成立時に1，不成立時に0が8ビット分埋められま

●表2．1　Bcc, DBcc で使われる cc の記述と意味

CCの記述	成　立　す　る　条　件
EQ	比較したとき等しかったことを表わす．一枚の演算結果がゼロになったことを検知するのにも使われる．
NE	比較した結果，等しくなかったことを示す．また，演算結果がゼロでないかどうかを検出するのにも用いられる．
HI	符号なし2進数として比較したとき，大きかったことを表わす．
CC	C（キャリー）フラグが0のとき．すなわち符号なし2進数として比較したとき，大きいかまたは等しかったことを示す． 単なる加算などで桁上がりが発生しなかったことを検出するのにも用いられる．
CS	Cフラグが1のときで，符号なし2進数として比較したとき，小さかったことを示す．演算で桁上がりが生じたことを検出する場合にも利用できる．
LS	符号なし2進数として比較した場合，小さいか等しかったことを表わす．
GT	符号付き2進数として比較したとき，大きかったことを表わす．
GE	符号付き2進数として比較したとき，大きいか等しかった場合を表わす．
LT	符号付き2進数として比較したとき，小さかったことを表わす．
LE	符号付き2進数として比較したとき，小さいか等しかったことを表わす．
PL	演算結果がプラスだったことを表わす．単に結果の最上位ビットが“0”であるかどうかを調べる場合にも使われる．
MI	演算結果がマイナスになったことを表わす．単に結果の最上位ビットが“1”であるかどうかを調べる場合にも利用される．
VS	演算の結果，オーバーフローが生じたことを表わす．
VC	演算の結果，オーバーフローが生じなかったことを表わす．
F（またはRA）	常に不成立．−1で抜ける機能だけが使われる．　(DB_{cc}，S_{cc})
T	常に成立．　(DB_{cc}，S_{cc})

す．この命令はバイト・サイズ専用です．なお，cc の指定は表2.1によって行ないます．

以上の各命令の実効によって，フラグは変化しません．

2-8　サブルーチンとスタック関係の命令

❖関連コマンド　　**jsr, bsr, rts, rtr, link, unlk, movem**

機械語レベルでのサブルーチンは，BASIC の GOSUB 命令的な感覚で実行できます．その場合，実効アドレスを PC に転送する方法でサブルーチンにジャンプする jsr と，PC 値に相対アドレスを加算してブランチする bsr のいずれかが使えます．これらの書き方は次のとおりです．

```
jsr  <ea>
bsr [.s]   <ラベル>
```

ここで，bsr のショート型は bra の場合と同じです．

これらの命令は，サブルーチンに飛ぶ前に，実行前の PC 値(言い換えると次の命令＝戻り先＝を示すアドレス)をスタックに記憶します(図2.8)．

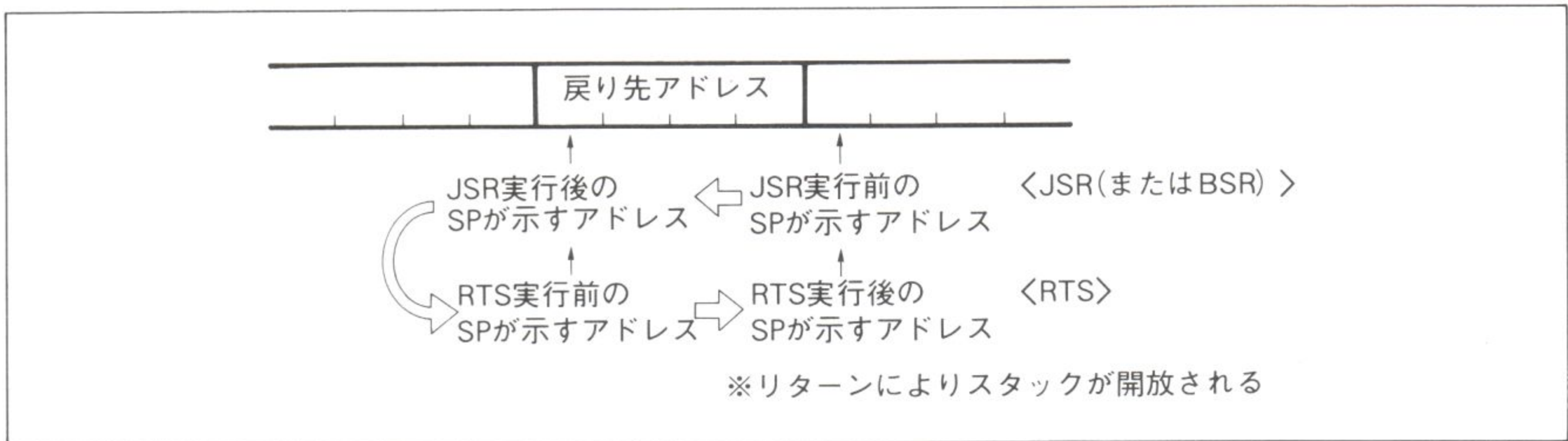
サブルーチンの側では，BASIC の RETURN に相当する命令として，rts などが使えます．rts はスタッ

●図2．8　JSRとRTSのスタック関係

戻り先アドレス

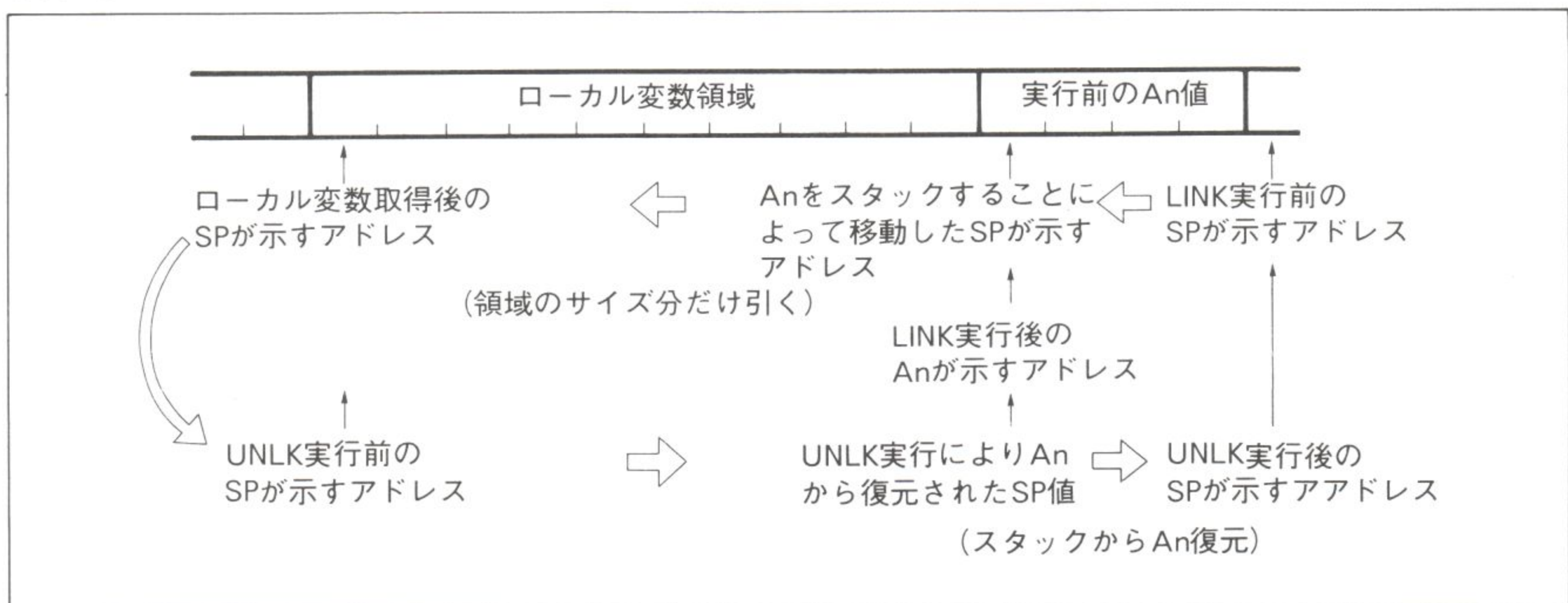
JSR実行後の
SPが示すアドレス
JSR実行前の
SPが示すアドレス
<JSR(またはBSR)>

RTS実行前の
SPが示すアドレス
RTS実行後の
SPが示すアドレス
<RTS>

※リターンによりスタックが開放される

●図2．9　LINKとUNLKのローカル変数領域の取得と開放

ローカル変数領域
実行前のAn値

ローカル変数取得後の
SPが示すアドレス
Anをスタックすることに
よって移動したSPが示す
アドレス
LINK実行前の
SPが示すアドレス
(領域のサイズ分だけ引く)
LINK実行後の
Anが示すアドレス

UNLK実行前の
SPが示すアドレス
UNLK実行によりAn
から復元されたSP値
UNLK実行後の
SPが示すアアドレス
(スタックからAn復元)

クにある戻り先アドレスを取り出しPCに転送するので，いわば戻り先に対するジャンプ命令のように働きます．この場合，スタックにあったアドレス値は実行時のPC値だったのですから，プログラムがどの位置にロードされようと，それに合った値がリレーされるので問題は起きません．

RETURNの代わりをするもう1つの命令は，**rtr**です．この命令はサブルーチンの入口で前もってCCR値をスタックするという前提で，リターンの際にCCR値も復元します．

rtsもrtrも，オペランドをもたないので，ニーモニックをこのとおり書くだけです．

サブルーチンではローカル変数を取り扱うことが少なくありませんが，このための領域の取得にはlink，開放には**unlk**が使われます．これらの命令はAn(アドレス・レジスタ)を1個占有し，図2.9のような動作を行ないます．

すなわちlink時は，Anの内容をスタックに退避したあとSP値を貰い受けます．こうしたあとでSPにローカル変数のサイズ(負数で指定)を加算すると，スタック内にローカル変数領域が確保されます．

unlkによりローカル変数を解放するときは，最初にAnの値がSPにコピーされ，この状態でスタックからAnの退避値を復元します．これによって，AnもSPも完全に元の値に戻ります．

以上の動作の間，占有されたAnを変化させることはunlkの処理を誤らせるため危険です．どうしてもAnも使いたいときは，Link実行後An値をローカル変数領域にでも退避させておき，unlk実行前に戻しておくような工夫が必要です．

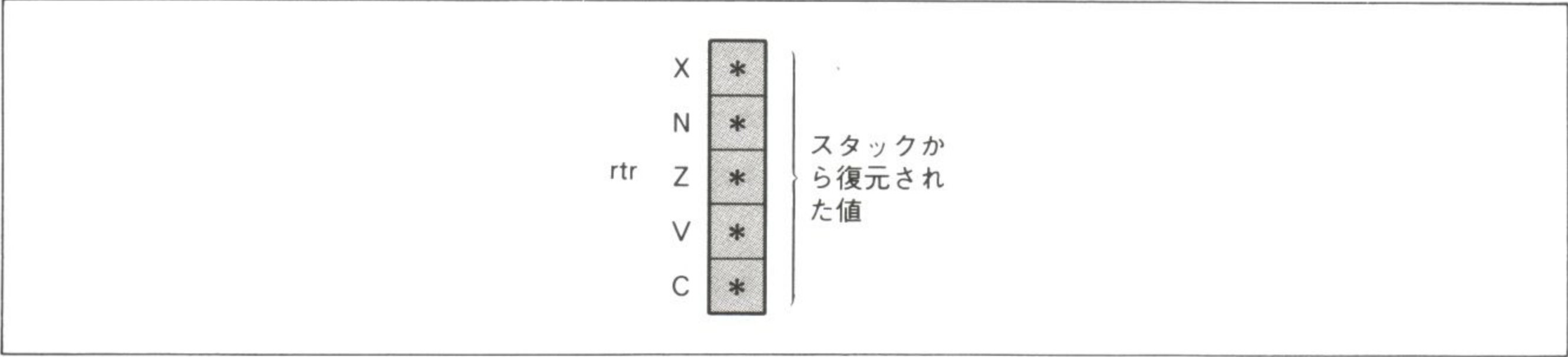
link，unlkの書き方は次のとおりです．

```
link    An,#−<ローカル変数領域サイズ>
unlk    An
```

サブルーチン実行時には，レジスタ値を変更しないで親ルーチンへ戻りたいケースが少なくありません．このようなときは，

●図2．10　rtr 命令実行後のフラグの変化

rtr	X	*	スタックから復元された値
	N	*	
	Z	*	
	V	*	
	C	*	

```
movem | .w |  <レジスタ・リスト>, −(SP)
      | .l |
```

命令を使って，レジスタ・リストで指定されたレジスタをスタックに退避します．反対にレジスタを復元するときは，

```
movem | .w |  (SP)+, <レジスタ・リスト>
      | .l |
```

でスタックから取り出します．movem は，メモリ側として上記以外の <ea> 記述をすることもできます．データ・サイズとしてワード(w)を指定するのは，特殊な場合(下位16ビットで足りるケース)に限られるので注意が必要です．また，その場合復元にあたって，上位16ビットのすべてに，下位16ビットの最上位(D_{15})ビットの値がコピーされます．

以上の各命令のうち，実行後フラグが変化するのは rtr のみで，図2.10のようになります．

2-9 その他の命令

❖関連コマンド　**trap, trapv, chk, rte, reset, stop**

ここでは，以上の分類のいずれにも該当しない命令について取り上げます．

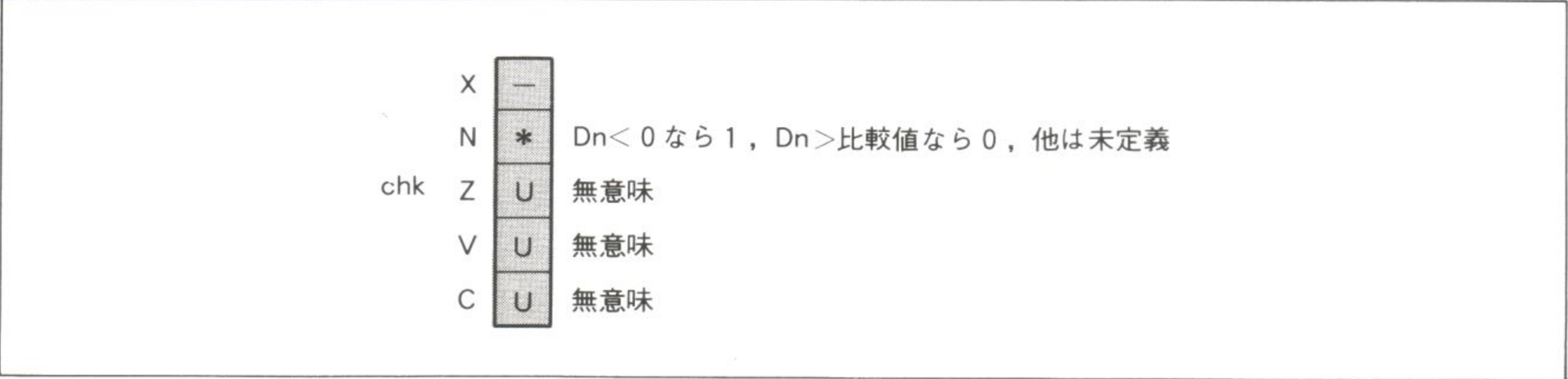
デバックなどによく使われる命令としては，

```
nop
```

があります．この命令は命令コードが存在するだけで何もしないのですが，不要になった命令の上に重ね書きしてつぶしたり，複数個並べて時間をつぶす(微調整的な感覚で使う)のに使われます．

割り込みに関するものには，

●図2．11　chk 命令実行後のフラグの変化

chk	X	−	
	N	*	Dn＜0なら1，Dn＞比較値なら0，他は未定義
	Z	U	無意味
	V	U	無意味
	C	U	無意味

```
trap # 〈ベクタ番号〉
trapv
chk  〈ea〉,Dn
```

があります．**trap** は0～15の**トラップ割り込み**を発生するもので，この命令が実行されると，該当番号のベクトルからアドレスが読み出され，そのアドレスが示す例外処理が起動されます．**trapv** は，**V フラグ**が ON の状態で実行すると**trapv 割り込み**を発生します．**chk** は Dn で指定されるデータ・レジスタの下位ワードが0より小さいか，ea で指定される対象より大きいときレジスタ境界チェックが働き，**chk 割り込み**を発生します．いずれも該当ベクトルにより割り込み処理ルーチンが起動されます．

割り込みルーチンはスーパーバイザ・モードで実行され，

```
rte
```

命令で終了します．この命令は特権命令のため，ユーザ・モードでは使えません．

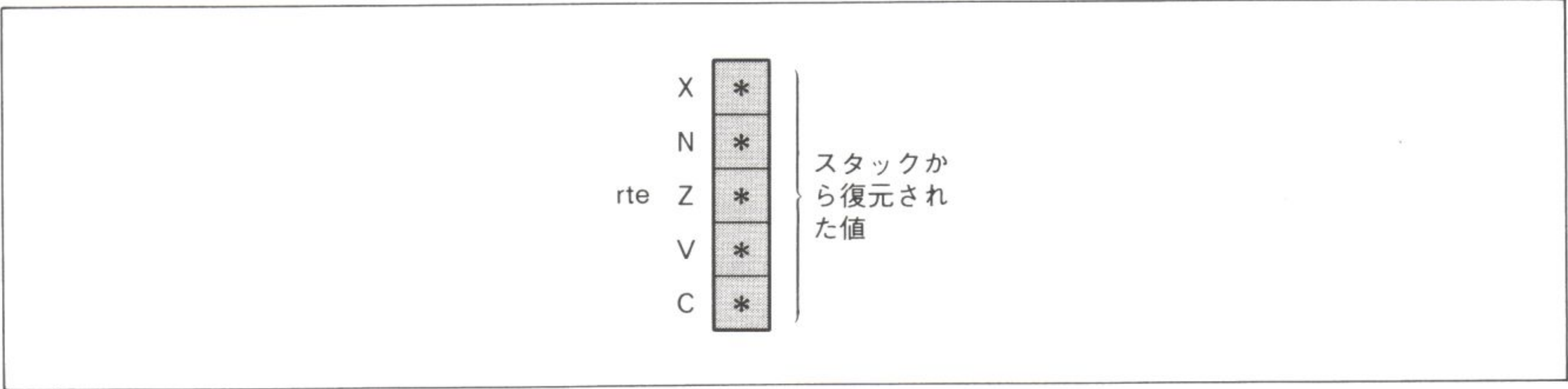
他の特権命令でまだ紹介していないものには，

```
reset
stop  #〈SR にセットするデータ〉
```

があります．**reset**はシステムのリセット線を働かせ，周辺デバイスをリセットするものです．**stop** は指定されたデータを SR に転送して停止する命令で，その後 SR にセットされた割り込みマスクより高い割り込みが発生すると終了します．

以上の命令のうち，chk，rte 命令を実行するとフラグは図2.11，図2.12のように変化します．stop の場合は，オペランドのデータがそのまま転送されます．

●図2．12　rte 命令実行後のフラグの変化

	X	*	スタックから復元された値
	N	*	
rte	Z	*	
	V	*	
	C	*	

第3章

アセンブラの実技

3-1 DOSコールの使い方

第4部でも述べたように，初めての言語を使うときは，何か適当なメッセージを表示してみるなど，動作を確認できるようにすることから着手するのが近道です．そこで，このためのテスト・プログラムを作成して走らせることを，筆者は「開通式」と呼んでいます．

ここではアセンブラ・プログラムの開通式のため，単にメッセージを出力するだけのプログラムを組んでみることにします．

アセンブラでは，入出力などHuman68Kモニタの機能を利用するときは，**ファンクション・コール**を使います．これはサブルーチンを使うのに似ていますが，普通のサブルーチンのようにリターン付きのジャンプやブランチを利用するのと違って，模擬命令によるエミュレータの手法で割り込みを使っています．したがって，個々のエミュレーション・ルーチンのアドレスを意識する必要もなければ，モニタのバージョン・アップによる変更の影響を受けることもありません．

ファンクション・コールのうちDOSコールを利用するときは，図3.1①のようにパラメータをスタックに入れ，その直後命令コード＄FFxxを実行(同図②)します．これから先はHuman68K内部の処理に移り，例外処理ルーチンが働きます(同図③)．この処理はRTEで終了(同図④)し，このときスタック・ポインタ(SP)は①直後の値に戻ります．したがって，ユーザ処理の側に戻った後は，パラメータのサイズだけスタックを開放しなければなりません．

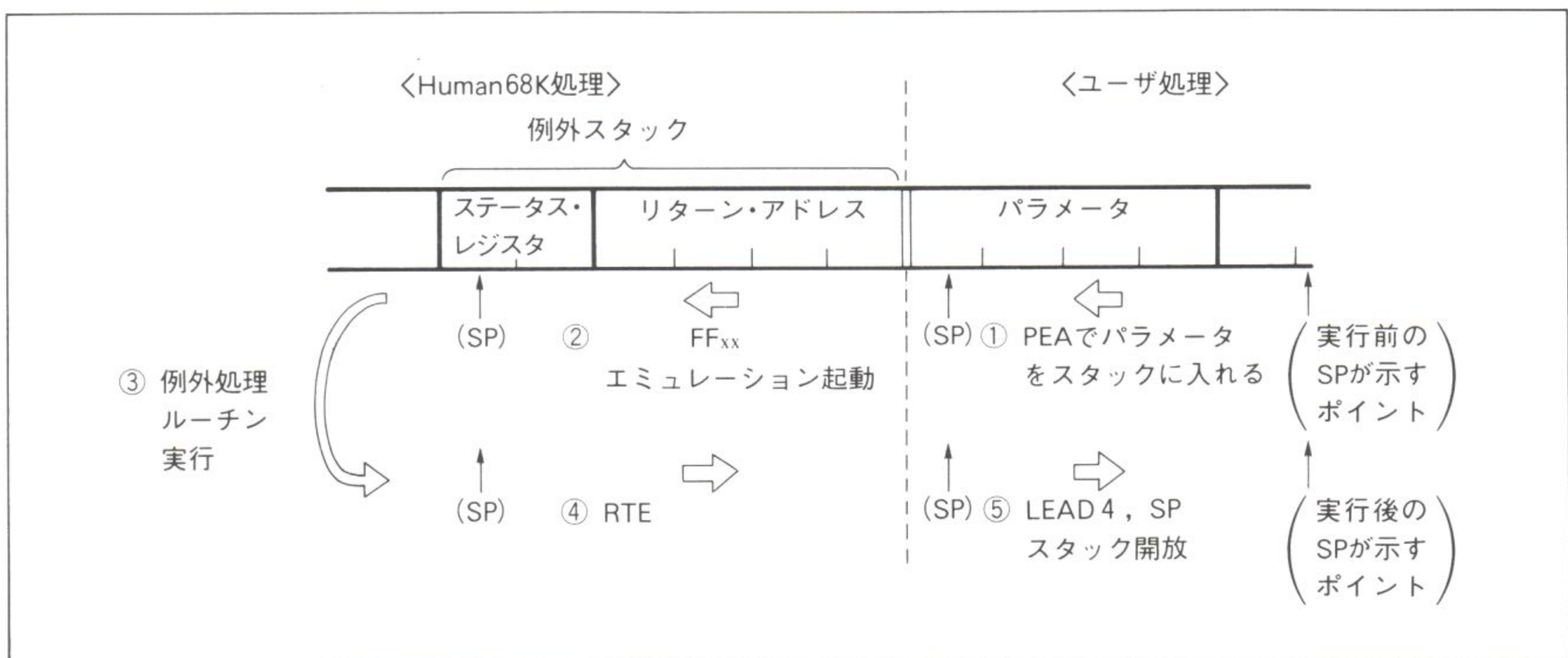
図3.2は，以上のことを確認するために作成したテスト・プログラムです．ここでは，Human68Kユーザーズマニュアルのファンクション・コールの説明に基づいて，_Print，_exitを使っています．ただし，マニュアルどおりに記述しても，インクルード・ファイルを使わないと_printなどのファンクション名が未定義になってしまうので，とりあえず16進記述としておきます．

プログラムは，単にメッセージを表示して，そのまま終了するだけです．_printで出力する文字列は，

本章ではアセンブラでプログラミングする際に使用する道具立て(インクルード・ファイルやマクロなど)について述べ，また入出力などに使用するDOSコール，IOCSコールの概要と使い方について説明します．

アセンブラには直接デバイスをコントロールできる「特技」があるので，アセンブラは，「原始的言語」と呼ばれながらもなかなか消滅しないのです．ここでは，デバイスを直接操作する例もあげているので，アセンブラ・プログラミング技術の「基礎体力」がついた読者は参考にしていろいろ試してみると面白いでしょう．

なお，個々の命令の使い方については，巻末のOS-9の記事に多くの参考例があるので，参考にしてください．

●図3．1 DOSコールとスタックの変化

●図3．2 テスト用のメッセージ表示プログラム (dispt)

```
 1 00000000                                    .text
 2 00000000                          begin:
 3 00000000 4879(02)00000000                   pea      message
 4 00000006 FF09                               dc.w     $ff09       _print
 5 00000008 588F                               addq.l   #4,sp
 6 0000000A FF00                               dc.w     $ff00       _exit
 7 0000000C
 8 0000000C                                    .data
 9 00000000 54657374206D6573   message         dc.b 'Test message',$0d,$0a,0
            736167650D0A00
10 0000000F                                    end begin
```

●図3．3　disptの実行結果

```
A>dispt
Test message
```

●表3．1　DOSコール一覧

コール	名　称	機　　　能
$FF01	getchar	キーボードの入力を待つ(エコーつき)
$FF02	putchar	文字コードにより文字を表示
$FF03	cominp	RS-232C回線から1バイト入力
$FF04	comout	RS-232C回線から1バイト出力
$FF05	prnout	プリンタに1文字出力
$FF06	inpout	キーの入出力
$FF07	inkey	キーボードから1文字入力(ブレーク・チェックなし)
$FF08	getc	キーボードから1文字入力(ブレーク・チェックあり)
$FF09	print	文字列を表示
$FF0A	gets	文字列を入力
$FF0B	keysns	キーの入力状態をチェック
$FF0C	kflush	バッファ・フラッシュつきキーボード入力
$FF10	consns	画面出力ができるかどうかをチェック
$FF11	prnsns	プリンタ出力ができるかどうかをチェック
$FF12	cinsns	RS-232C回線から入力可・不可のチェック
$FF13	coutsns	RS-232C回線への出力の可・不可のチェック
$FF18	hendsp	漢字変換行をコントロール
$FF1A	getss	文字列を入力
$FF22	knjctrl	かな漢字変換ファンクション・コール呼び出し
$FF23	conctrl	コンソールに直接出力
$FF24	keyctrl	コンソールから直接入力
$FF44	ioctrl	デバイス・ドライバのIOCTLによる入出力
$FF48	malloc	メモリを確保
$FF49	mfree	メモリ・ブロックを開放
$FF4A	setblock	メモリ・ブロックを変更
$FF00	exit	プログラムを終了
$FF26	pspset	バッファに終了情報を書き込む
$FF31	keeppr	常駐終了
$FF4C	exit2	終了コードを指定して終了
$FF4D	wait	プロセスの終了コードを得る
$FF50	setpdb	管理プロセスを移す
$FF51	getpdb	現在のプロセス情報を得る
$FF52	setenv	環境変数を設定
$FF53	getenv	環境変数の内容を得る
$FF1B	fgetc	ファイル・ハンドルから1バイト入力
$FF1C	fgets	ファイル・ハンドルから文字列を入力
$FF1D	fputc	ファイル・ハンドルに1バイト出力
$FF1E	fputs	ファイル・ハンドルに文字列を出力
$FF1F	allclose	ファイル・ハンドルをクローズ
$FF29	namests	ファイル情報をバッファに展開
$FF2F	dup0	ファイル・ハンドルを強制複写
$FF37	nameck	ファイル情報の展開
$FF3C	creat	ファイルを作成
$FF3D	open	ファイルをオープン
$FF3E	close	ファイル・ハンドルをクローズ
$FF3F	read	ファイルから読み込む
$FF40	write	ファイルへ書き出す
$FF41	delete	ファイルを削除
$FF42	seek	ファイル・ポインタの移動
$FF45	dup	ファイル・ハンドルを複写
$FF46	dup2	ファイル・ハンドルを強制複写
$FFF3	diskred	ブロック・デバイスから直接読み込む
$FFF4	diskwrt	ブロック・デバイスに直接書き込む
$FF39	mkdir	ディレクトリを作成
$FF3A	rmdir	ディレクトリを削除
$FF3B	chdir	カレント・ディレクトリを変更
$FF43	chmod	ファイルの属性を変更

コール	名 称	機　　能
$FF47	curdir	パス名を設定
$FF4E	files	ファイル情報を検索(最初のファイル)
$FF4F	nfiles	ファイル情報を検索
$FF56	rename	ファイル名を変更(移動)
$FF57	filedate	ファイルの日付／時間の読み出し／設定
$FF0D	fflush	ディスクをリセット
$FF0E	chgdrv	カレント・ドライブを指定
$FF0F	drvctrl	ドライブの状態をチェック／設定
$FF17	fatchk	ドライブのセクタが連続しているかどうかをチェック
$FF19	curdrv	カレント・ドライブを返す
$FF20	super	スーパーバイザ・モード／ユーザ・モードの切り換え
$FF21	fnckey	再定義可能なキーの読み出し／設定
$FF25	intvcs	INTNOで指定するベクタにアドレスを設定
$FF27	gettim2	現在の時刻を得る
$FF28	settim2	時刻を設定
$FF2A	getdate	日付を得る
$FF2B	setdate	日付を設定
$FF2C	gettime	時刻を得る
$FF2D	settime	時刻を設定
$FF2E	verify	ベリファイ・フラグを設定
$FF30	vernum	OSのバージョン番号を得る
$FF32	getdpb	指定装置番号のドライブ・パラメータ・ブロックのコピー
$FF33	breakck	ブレーク・チェックを設定
$FF34	drvxchg	ドライブを入れ換える
$FF35	intvcg	INTNOで指定するベクタ値を得る
$FF36	dskfre	ディスクの残り容量を得る
$FF54	verifyg	ベリファイ・フラグの設定状況を得る

＄0のヌル文字で終わっていなければならないので，全体の最後に0を付けるのと，改行のため＄0D，＄0Aの制御コードを付けているのがBASICやCで取り上げたサンプルと異なるところです．

すなわち，アセンブラのほうが，よりナマに近いデータを扱うので，舞台裏までさらけ出してプログラミングする感覚となります．このことがハードウェアの理解を助け，パソコンをより高度に使う技術を磨くのに役立つことはいうまでもありません．

上述のプログラムのテスト結果は，図3.3のとおりです．

DOSコールの一覧を表3.1に示します．

3-2　作っておいて役に立つインクルード・ファイル

プログラムをモジュール化する際，一般にはオブジェクト・ファイル・レベルでの分割を考えます．

しかし，もともと1つだったものを分割するということは，分割後の各モジュールから統一的に参照される情報をもつことにもつながります．これらの情報は，各モジュールのソースを記述する際に個々に与えることもできますが，そのままでは同じものをあちこちに書くという面倒さや，記述ミスによる不統一などの問題が生じやすいため，あまり感心できません．

そこで，実際は**インクルード・ファイル**を利用し，共通に参照する内容をすべてこの中から得るようにしています．こうすればモジュール間の不統一はなくなります．さらにインクルード・ファイルの内容に修正が発生した場合は，関連モジュールを再アセンブルするだけで全モジュールの更新ができます．このようなケースでは，インクルード・ファイルなしで全モジュールを手修正することを考えれば，どれだけ作業の改善につながるか想像できることと思います．

以上で述べたことは，インクルード・ファイルの一般的な使い方ですが，見方を変えれば別な目的での利用も可能です．要は，アセンブラ記述でinclude擬似命令を使って，該当部分に他のソース・ファイルの内容を挿入するのですから，命令群などを任意のブロックに分けておいて，必要に応じて呼び込むようにすれば，ソース・ファイル・レベルでのモジュール化ができます．

●図3．4　ファンクション・コールのためのインクルード・ファイル

```
 .nlist
_exit     equ   $ff00
_getchar  equ   $ff01
_putchr   equ   $ff02
_cominp   equ   $ff03
_comout   equ   $ff04
_prout    equ   $ff05
_inpout   equ   $ff06
_inkry    equ   $ff07
_getc     equ   $ff08
_print    equ   $ff09
_gets     equ   $ff0a
_keysns   equ   $ff0b
_kflush   equ   $ff0c
_fflush   equ   $ff0d
_chgdrv   equ   $ff0e
_drvctrl  equ   $ff0f
_consns   equ   $ff10
_prnsns   equ   $ff11
_cinsns   equ   $ff12
_coutsns  equ   $ff13
_fatchk   equ   $ff17
_curdrv   equ   $ff19
_getss    equ   $ff1a
_super    equ   $ff20
_intvcs   equ   $ff25
_pspset   equ   $ff26
_gettim2  equ   $ff27
_settim2  equ   $ff28
_namests  equ   $ff29
_getdate  equ   $ff2a
_setdate  equ   $ff2b
_gettime  equ   $ff2c
_settime  equ   $ff2d
_verify   equ   $ff2e
_dup0     equ   $ff2f
_vernum   equ   $ff30
_keeppr   equ   $ff31
_getdpd   equ   $ff32
_breakck  equ   $ff33
_intvcg   equ   $ff35
_dskfre   equ   $ff36
_mkdir    equ   $ff39
_rmdir    equ   $ff3a
_chdir    equ   $ff3b
_creat    equ   $ff3c
_open     equ   $ff3d
_close    equ   $ff3e
_read     equ   $ff3f
_write    equ   $ff40
_delete   equ   $ff41
_seek     equ   $ff42
_chmod    equ   $ff43
_ioctrl   equ   $ff44
_dup      equ   $ff45
_dup2     equ   $ff46
_cutdir   equ   $ff47
_malloc   equ   $ff48
_mfree    equ   $ff49
_setblock equ   $ff4a
_exec     equ   $ff4b
_exit2    equ   $ff4c
_wait     equ   $ff4d
_files    equ   $ff4e
_nfiles   equ   $ff4f
_setpdb   equ   $ff50
_getpdb   equ   $ff51
_verifyg  equ   $ff54
_rename   equ   $ff56
_filrdate equ   $ff57
 .list
```

●図３．５　dispt をインクルード・ファイルを使うように書き直したもの

```
   1 00000000                                 include Fcall
   1 00000000                        .list
   1 00000000
   2 00000000                                 .text
   3 00000000                       begin:
   4 00000000 4879(02)00000000                pea     message
   5 00000006 FF09                            dc.w    _print
   6 00000008 588F                            addq.l  #4,sp
   7 0000000A FF00                            dc.w    _exit
   8 0000000C
   9 0000000C                                 .data
  10 00000000 54657374206D6573      message dc.b 'Test message',$0d,$0a,0
              736167650D0A00
  11 0000000F                                 end begin
```

さて，Ｃコンパイラ・セットを入手する以前にインクルード・ファイルの利用について筆者が最も切実に感じたのは，次のような場面です．

X68000のアセンブラでファンクション・コールする際，たとえば，

```
dc.w   _exit
```

のように記述するのですが，いきなりこのように書いてもエラーになってしまいます．実際は，前もって

```
_exit   equ   $ff00
```

を定義しておかなければならないのです．

このようなシンボル名は，プログラムを作成する際に必ずといってよいほど登場するものですが，そのままでは毎回上述のように定義した上で参照せざるを得ません．もちろん前節のように，ファンクション・コール時に

```
dc.w   $ff00
```

と書けば，別途定義する必要もありませんが，これではコメントを付けないとプログラムがわかりにくいものになってしまいます．

そこで筆者が最初に作ったインクルード・ファイルでは，これらのファンクション・コールのシンボル名をずらりと並べて定義しました(図3.4)．一度このファイルを作っておけば，あとはDOSコールを必要とするモジュールの先頭部分で，

```
include   Fcall
```

と記述するだけでシンボルの定義が代行されます．

ところが，手間ヒマかけてインクルード・ファイルを作ったにもかかわらず，Ｃコンパイラ・セットが発売され，入手できるようになってからはこのファイルは不要になりました．このセットにはDOSコールのための定義ファイルが付いてきたのです．この場合の参照記述は，

```
include   ¥include¥doscall.mac
```

とします．

このセットをもっていない読者には，図3.4の例に習ってインクルード・ファイルを作っておくことをおすすめします．

このインクルード・ファイルを使って，前節のプログラムを書き直すと，図3.5のようになります．プログラムの先頭で，必ずincludeを入れるのを忘れないようにくれぐれも注意してください．

3-3 マクロの定義とその使い方

プログラムの中には,DOSコール手続きなど決まりきった命令の並びとなるところが少なくありません.このような記述は，マクロ命令を使用することによって大幅に簡略化できます.

マクロの機能は，定義する側と参照する側の両方から見なければなりません．1つのマクロの定義は，macro擬似命令に始まり，endm擬似命令で終わります．そして，macro擬似命令は，次の形式で定義されます.

<ラベル>　macro　[<パラメータ・リスト>]

ここで，ラベルはマクロを呼び出す際の名前を書き，パラメータ・リストは，もしパラメータをもつならば，第1パラメータから順に“,”で区切って並べます.

あとはendmまでの間に任意の命令行(擬似命令も可)を並べればよいのですが，そのうちパラメータを参照するものはパラメータ・リスト中の該当する変数名を使用できます.

条件付きアセンブルの擬似命令との組み合わせによって,任意の行からマクロの展開を終了させる(endmまでの行を無効にする)場合は,

exitm

擬似命令を使用します.

また，マクロ記述において，そのマクロ内だけで使用するシンボル名を定義したいときは,

●図3．6　マクロを使用しないときのプログラム

```
  1 00000000                         *
  2 00000000                         *      None Macro
  3 00000000                         *
  4 00000000
  5 00000000                                include ¥include¥doscall.mac
  5 00000000                                .list
  6 00000000
  7 00000000                         start:
  8 00000000 4879(01)0000000C                pea      text_line      }
  9 00000006 FF09                            dc.w     _PRINT         }
 10 00000008 588F                            addq.l   #4,sp          } 処理の記述
 11 0000000A FF00                            dc.w     _EXIT          }
 12 0000000C                         *
 13 0000000C 54657374206F6B0D        text_line dc.b 'Test ok',$0d,$0a,0
             0A00
 14 00000016                                end      start
```

●図3．7　マクロ・ファイルの定義内容

```
         .nlist

exit     Macro
         dc.w     _EXIT
         Endm

print    Macro    string
         pea      string
         dc.w     _PRINT
         addq.l   #4,sp
         Endm

         .list
         include ¥include¥doscall.mac
```

local　〈シンボル名〉｛，〈シンボル名〉｝

擬似命令を使用します．このような要求は，マクロ内でブランチする場合，ブランチ先のラベルとして利用するなどのケースで生じ，局所に限定しないとラベルが重複する恐れがあるとき用いられます．

マクロの終了は，

endm

擬似命令を使います．この擬似命令の次には，また別なマクロの定義を並べることができます．

一方，マクロを参照する側では，最初にinclude擬似命令でマクロ定義の記述されたファイルを読み込みます．そうしておいて，必要な部分で，

〈マクロ名〉　［〈パラメータ〉｛，〈パラメータ〉｝］

形式で参照するのです．このとき該当マクロの展開が行なわれ，パラメータがあるときは，記述順にパラメータ・リストと一致する位置に代入されます．

図3.6はマクロを使用しないプログラムの例で，ここでは単に，

Test ok

というメッセージを出しているだけです．

このプログラムで_PRINTと_EXITのDOSコールを行なっていますが，前者は手続きに3ステップ必要で，メッセージ出力はあちこちのプログラムでよく行なうことだけに，マクロ化してしまいたいところです．

そこで図3.7のように，文字ストリングのアドレスをパラメータとしたマクロを定義し，ついでに_EXITについても登録してしまうことにしました．登録内容については，includeで参照されるときリスティングされると邪魔になるので，.nlistを使って抑制しています．また，マクロ定義からdoscall.macを参照しているので，このためのinclude擬似命令も含めました．

このマクロを利用して組み直したプログラムは，図3.8のように処理記述が非常にすっきりとしたものに

●図3．8　図3．6のプログラムをマクロを使って書き直したもの

```
 1 00000000                          *
 2 00000000                          *      Test of Macro
 3 00000000                          *
 4 00000000
 5 00000000                                  include Macro
 5 00000000                                  .list
 5 00000000                                  include ¥include¥doscall.mac
 5 00000000                                  .list
 5 00000000
 6 00000000
 7 00000000                          start:
 8 00000000                                  print    text_line        } 処理の記述
 9 0000000A                                  exit
10 0000000C                          *
11 0000000C 54657374206F6B0D         text_line dc.b 'Test ok',$0d,$0a,0
            0A00
12 00000016                                  end      start
```

●図3．9　プログラムの実行結果

```
A>mact
Test ok
```

なりました．ここでは，機械語命令に該当するものはひとつもありません．

このプログラムを“mact”と命名し，テストした結果は図3.9のとおりです．mact コマンド(プログラム)を実行すると，内部で定義された文字列が正しく出力されたことがわかります．

なお，ここではインクルード・ファイルとして自作のものは使わず，C コンパイラ・セットのものを利用しました．また Human68K の Ver.1.00ではマクロを使うとアセンブラがクラッシュしたので，C コンパイラ・セットのもの(Ver.1.01)を使いました．

3-4 ヒストリ・ファイルを加工するプログラム

アセンブラ・ブログラミングの便利な道具について説明したあとは，少し役に立ちそうなプログラムに取り組んでみます．

ここで取り上げるサンプルは，ヒストリ・ファイルの内容を加工するもので，his コマンドでセーブしたヒストリ・ファイルに行番号が含まれているとき，これをカットしたファイルを作成しようというのがねらいです．his コマンドでディスク・ファイルにリダイレクトするまではよいのですが，／B スイッチを忘れることは意外にあります．このようなときヒストリ・ファイルを作り直せば以前とは少し違った内容になってしまいます．そこで，ヒストリ・ファイルはそのままで，あたかも／B スイッチを使ったように変換するプログラムが欲しくなるのです．

このためプログラムは，ヒストリ・ファイルを読み，コマンド行の先頭から 6 文字を削除するだけですから，理論的には簡単です．しかしここでは少し遠まわりして，Human68K のアセンブラで用意されているディスク読み込み用のDOSコール_**read** を使ってみることにします．_read はブロック単位で入力を行なうようになっているので，これを使う限りはプログラマが一つ一つのコマンド行を取り出さなければなりません．このため，プログラムのサンプルとしては参考になりそうな場面がたくさん出てきそうです．

図3.10のプログラムは，元のヒストリ・ファイルを標準入力から読み込んで，行番号除去後のものを標

●図 3．10　ヒストリ・ファイルから行番号を除去するプログラム（hisin. s)

```
    1 00000000                                    include Fcall
    1 00000000                            .list
    1 00000000
    2 00000000
    3 00000000                                    .text
    4 00000000                          *
    5 00000000                          *      Main routine
    6 00000000                          *
    7 00000000                          start:
    8 00000000 612E_00000030                      bsr     readblock
    9 00000002 247C(02)0000000C                   move.l  #recbuf,a2
   10 00000008 614A_00000054                      bsr     copyrec
   11 0000000A                          loop:
   12 0000000A 615E_0000006A                      bsr     getrec
   13 0000000C 6720_0000002E                      beq.s   eop
   14 0000000E 2239(02)00000008                   move.l  recsize,d1
   15 00000014 5D81                               sub.l   #6,d1
   16 00000016 2F01                               move.l  d1,-(sp)
   17 00000018 4879(02)00000012                   pea     recbuf+6
   18 0000001E 3F39(02)00000002                   move.w  opath,-(sp)
   19 00000024 FF40                               dc.w    _write
   20 00000026 4FEF000A                           lea     10(sp),sp
   21 0000002A 6178_000000A4                      bsr     siftrec
   22 0000002C 60DC_0000000A                      bra.s   loop
   23 0000002E
   24 0000002E                          eop:
   25 0000002E FF00                               dc.w    _exit
   26 00000030
   27 00000030                          *
   28 00000030                          *     Read data block
   29 00000030                          *
   30 00000030                          readblock:
```

```
 31 00000030 2F3C00000100              move.l  #256,-(sp)
 32 00000036 4879(02)0000010D          pea     blkbuf
 33 0000003C 3F39(02)00000000          move.w  ipath,-(sp)
 34 00000042 FF3F                      dc.w    _read
 35 00000044 4FEF000A                  lea     10(sp),sp
 36 00000048 4A80                      tst.l   d0
 37 0000004A 6706_00000052             beq.s   rbEx
 38 0000004C 23C0(02)00000004          move.l  d0,blksize
 39 00000052                  rbEx:
 40 00000052 4E75                      rts
 41 00000054
 42 00000054                  *
 43 00000054                  *     Copy data line
 44 00000054                  *
 45 00000054                  copyrec:
 46 00000054 227C(02)0000010D          move.l  #blkbuf,a1
 47 0000005A                  clp:
 48 0000005A 14D9                      move.b  (a1)+,(a2)+
 49 0000005C 53B9(02)00000004          sub.l   #1,blksize
 50 00000062 66F6_0000005A             bne.s   clp
 51 00000064 14BC00FF                  move.b  #$ff,(a2)
 52 00000068 4E75                      rts
 53 0000006A
 54 0000006A                  *
 55 0000006A                  *     Get record
 56 0000006A                  *
 57 0000006A                  getrec:
 58 0000006A 42B9(02)00000008          clr.l   recsize
 59 00000070 247C(02)0000000C          move.l  #recbuf,a2
 60 00000076                  glp1:
 61 00000076 1212                      move.b  (a2),d1
 62 00000078 B23C000A                  cmp.b   #$0a,d1
 63 0000007C 671C_0000009A             beq.s   egr
 64 0000007E B23C00FF                  cmp.b   #$ff,d1
 65 00000082 670A_0000008E             beq.s   apnd
 66 00000084 528A                      inc.l   a2
 67 00000086 52B9(02)00000008          inc.l   recsize
 68 0000008C 60E8_00000076             bra.s   glp1
 69 0000008E                  apnd:
 70 0000008E 264A                      move.l  a2,a3
 71 00000090 619E_00000030             bsr     readblock
 72 00000092 670E_000000A2             beq.s   grEx
 73 00000094 61BE_00000054             bsr     copyrec
 74 00000096 244B                      move.l  a3,a2
 75 00000098 60DC_00000076             bra.s   glp1
 76 0000009A                  egr:
 77 0000009A 528A                      inc.l   a2
 78 0000009C 52B9(02)00000008          inc.l   recsize
 79 000000A2                  grEx:
 80 000000A2 4E75                      rts
 81 000000A4
 82 000000A4                  *
 83 000000A4                  *     Sift to next line
 84 000000A4                  *
 85 000000A4                  siftrec:
 86 000000A4 227C(02)0000000C          move.l  #recbuf,a1
 87 000000AA                  slp:
 88 000000AA 121A                      move.b  (a2)+,d1
 89 000000AC 12C1                      move.b  d1,(a1)+
 90 000000AE B23C00FF                  cmp.b   #$ff,d1
 91 000000B2 66F6_000000AA             bne.s   slp
 92 000000B4 224A                      move.l  a2,a1
 93 000000B6 4E75                      rts
 94 000000B8
 95 000000B8                  *
 96 000000B8                  *     Data area
 97 000000B8                  *
 98 000000B8                           .data
 99 00000000 0000             ipath    dc.w    0
100 00000002 0001             opath    dc.w    1
101 00000004                  blksize  ds.l    1
102 00000008                  recsize  ds.l    1
103 0000000C                  recbuf   ds.b    257
104 0000010D                  blkbuf   ds.b    256
```

●図3．11　テストに使ったヒストリ・ファイルの例（hisfile）

```
00005:dir
00004:del hisbat.bak
00003:cd asm
00002:ed hiain.s
00001:ed hisin.s
00000:at
```

●図3．12　hisinの実行結果（A>hisin <hisfile> hisfile.batにより実行した）

```
dir
del hisbat.bak
cd asm
ed hiain.s
ed hisin.s
at
```

準出力に送り出すものです．

ここでは，入力ファイルの読み出し結果はblkbufに入りますが，その内容はrecbufに追加され，そこでコマンド行1行ずつの加工出力を行ないます．そして出力した部分の内容はシフトされ，後続のコマンドがrecbuf内でどんどん繰り上がっていきます．

recbufの末尾には＄FFのダミー・データが付いており，これを含むコマンド行を検出すると，次のブロックが入力され，recbufに追加されます．こうして，ブロック間をまたがるコマンド行も支障のないように接続します．

入力ファイルの終わりの検出は，_read実行後のD_0の値を参照し，0ならば終了とみなします．プログラムでは，readblockサブルーチンでこれを行なっており，このときZフラグがセットされます．一方，終了でないときはZフラグがクリアされたままrtsに達するので，ファイルが終了したかどうかはこのサブルーチンを使っている親ルーチンでも簡単にわかります．たとえばgetrecサブルーチンでは，readblockを実行したのちbeq命令で終了時のバイパスを行なうようになっています．このときもZフラグは保存されるので，getrecを利用しているメイン・ルーテンでも参照しています．

なお，このプログラムは空のファイルを入力しないという前提で組んでいます．もし空のファイルにも対応できる(誤動作しない)ようにするためには，startでreadblockサブルーチン実行直後にZフラグを参照してeofにバイパスするように

```
beq.s    eof
```

を追加しておくのがよいでしょう．

このプログラムに図3.11のヒストリ・ファイルを入力してテストした結果は，図3.12のとおりです．

3-5 IOCSコール

DOSコールに役割りとしては似ていますが，異なった動作体系をもっているのが**IOCSコール**です．

IOCSコールは，trap割り込みの15番を利用するもので，パラメータはレジスタによって受け渡しします．そのうちD_0は，ロング・ワードで番号を指定するのに使われます．

IOCSコールの種類は表3.2のとおりで，非常に数が多いのが特徴です．

アセンブラでは直接デバイスをコントロールすることができますが，プログラムが面倒なので，DOSコールやIOCSコールで間に合わせるほうが賢明です．いわば「関数言語」と呼ばれるCのような感覚です．

また，IOCSコールの中にはタイマ・セットなどといった使って便利な機能も用意されているので，DOS

●表３．２　IOCSコール一覧①

参考(DO.L)	名称	機能
$00	B_KEYINP	１キーデータ入力
$01	B_KEYSNS	キーバッファ・チェック
$02	B_SFTSNS	シフトキー状態チェック
$03	KEY_INIT	キーの初期化
$04	B_BITSNS	キー状態チェック
$0C	TVCTRL	テレビ・コントロール
$0D	LEDMOD	LEDモードの指定
$0E	TGUSEND	画面の使用状態をOSに知らせる
$0F	DEFCHR	外字定義
$10	CRTMOD	CRTモードの指定
$11	CONTRAST	コントラストの設定
$12	HSVTORGB	HSV→RGB変換
$13	TPALET	テキスト・パレットの指定
$15	TCOLOR	テキスト・プレーン指定
$16	FNTADR	漢字フォント・アドレスを求める
$17	VRAMGET	VRAMからの読み込み
$18	VRAMPUT	VRAMへの書き込み
$19	FNTGET	フォント・データ読み込み
$1A	TEXTGET	パターン・ゲット
$1B	TEXTPUT	パターン・プット
$1C	CLIPPUT	パターン・プット(クリッピング処理をする)
$1D	SCROLL	表示範囲の設定／チェック
$1E	B_CURON	カーソル表示
$1F	B_CUROFF	カーソル非表示
$20	B_PUTC	１文字出力(漢字可)
$21	B_PRINT	文字列出力
$22	B_COLOR	文字属性設定
$23	B_LOCATE	カーソル移動(座標指定)
$24	B_DOWN_S	カーソル移動１行下(最下行でスクロール)
$25	B_UP_S	カーソル移動１行上(先頭行でスクロール)
$26	B_UP	カーソル移動n行上
$27	B_DOWN	カーソル移動n行下
$28	B_RIGHT	カーソル移動n桁右
$29	B_LEFT	カーソル移動n桁左
$2A	B_CLR_ST	画面の複数行にわたって消去
$2B	B_ERA_ST	画面の現在行の複数桁消去
$2C	B_INS	インサート(行単位)
$2D	B_DEL	デリート(行単位)
$2E	B_CONSOL	表示範囲の指定
$2F	B_PUTMES	座標指定文字列出力
$30	SET232C	RS-232Cパラメータ設定
$31	LOF232C	RS-232C受信バッファのデータ数を求める
$32	INP232C	RS-232C受信データを得る
$33	ISN232C	RS-232C受信データをチェック
$34	OSNS232C	RS-232Cが送信可能かどうかをチェック
$35	OUT232C	RS-232Cに送信を行なう
$3B	JOYGET	ジョイスティック・ポートのデータを読み込む
$3C	INIT_PRN	プリンタ・ポートの初期化
$3D	SNSPRN	プリンタ出力可能かどうかをチェック
$3E	OUTLPT	プリンタ直接出力
$3F	OUTPRN	プリンタ出力(漢字も可能)
$40	B_SEEK	ディスクのシーク
$41	B_VERIFY	ディスク・データのベリファイ
$42	B_READDI	診断のためのディスクからの読み取り
$43	B_DSKINI	デイスクの初期化
$44	B_DRVSNS	ディスクの状態のチェック
$45	B_WRITE	ディスクへの書き込み
$46	B_READ	ディスクからの読み込み
$47	B_RECARI	ディスクのリキャリブレイト
$48	B_ASSIGN	代替トラックの設定(HD)
$49	B_WRITED	破損データ書き込み(2HD)
$4A	B_READID	ID読み込み(2HD)
$4B	B_BADFMT	破損トラックを使用不能にする(HD)
$4C	B_READD	破損データの読み込み(2HD)

●表3．2　IOCSコール一覧②

参考(DO.L)	名称	機能
$4D	B_FORMAT	ディスクのフォーマット
$4E	B_DRVCHK	ドライブ状態設定(2HD)
$4F	B_EJECT	イジェクト(2HD)／シッピング(HD)
$50	DATEBCD	日付データの変換　バイナリ→BCD
$51	DATESET	日付の設定
$52	TIMEBCD	時刻データの変換　バイナリ→BCD
$53	TIMESET	時刻の設定
$54	DATEGET	日付の読み込み(BCD)
$55	DATEBIN	日付データの変換　BCD→バイナリ
$56	TIMEGET	時刻の読み込み(BCD)
$57	TIMEBIN	時刻データの変換　BCD→バイナリ
$58	DATECNV	日付データの変換　文字列→バイナリ
$59	TIMECNV	時刻データの変換　文字列→バイナリ
$5A	DATEASC	日付データの変換　バイナリ→文字列
$5B	TIMEASC	時刻データの変換　バイナリ→文字列
$5C	DAYASC	曜日データの変換　バイナリ→文字列
$5D	ALARMMOD	アラームの禁止／許可設定・チェック
$5E	ALARMSET	アラームの時間と処理アドレス設定
$5F	ALARMGET	アラームの時間と処理アドレス・チェック
$60	ADPCMOUT	ADPCMのデータ出力(ブロック)
$61	ADPCMINP	ADPCMのデータ入力(ブロック)
$62	ADPCMAOT	ADPCMのデータ出力(アレイ・チェーン)
$63	ADPCMAIN	ADPCMのデータ入力(アレイ・チェーン)
$64	ADPCMLOT	ADPCMのデータ出力(リンク・チェーン)
$65	ADPCMLIN	ADPCMのデータ入力(リンク・チェーン)
$66	ADPCMSNS	ADPCMの状態チェック
$67	ADPCMMOD	ADPCMの実行制御
$68	OPMSET	OPM(FM音源)のレジスタ設定
$69	OPMSNS	OPM(FM音源)の状態チェック
$6A	OPMINTST	OPM(FM音源)の割り込みの設定
$6B	TIMERDST	MFPのTIMER-Dによる割り込み制御
$6C	VDISPST	垂直同期による割り込み制御
$6D	CRTCRAS	ラスター走査による割り込み制御
$6E	HSYNCST	水平同期信号立ち下がりによる割り込み制御
$6F	PRNINTST	プリンタによる割り込み制御
$70	MS_INIT	マウスの初期化
$71	MS_CURON	マウス・カーソルの表示
$72	MS_CUROF	マウス・カーソルの非表示
$73	MS_STAT	マウス・カーソルの表示モード・チェック
$74	MS_GETDT	マウスの移動量／ボタンの状態チェック
$75	MS_CURGT	マウスの現在座標チェック
$76	MS_CURST	マウスの座標指定
$77	MS_LIMIT	マウスの移動範囲指定
$78	MS_OFFTM	マウス・ボタンを離すまでの時間のチェック
$79	MS_ONTM	マウス・ボタンを押すまでの時間のチェック
$7A	MS_PATST	マウス・カーソル・パターンを定義
$7B	MS_SEL	マウス・カーソル・パターンを指定
$7C	MS_SEL2	マウス・カーソル・アニメーション
$7D	SKEY_MOD	ソフト・キーボードの制御
$7E	DENSNS	電卓の制御
$7F	ONTIME	電源起動後の経過時間を返す
$80	B_INTVCS	ベクタの設定
$81	B_SUPER	スーパーバイザ／ユーザ・モードの切り換え
$82	B_BPEEK	指定アドレスから1バイト読み込み
$83	B_WPEEK	指定アドレスから1ワード読み込み
$84	B_LPEEK	指定アドレスから1ロング・ワード読み込み
$85	B_MEMSTR	指定アドレスから複数バイト読み込み
$86	B_BPOKE	指定アドレスへ1バイト書き込み
$87	B_WPOKE	指定アドレスへ1ワード書き込み
$88	B_LPOKE	指定アドレス1ロング・ワード書き込み
$89	B_MEMSET	指定アドレスへ複数バイト書き込み
$8A	DMAMOVE	DMA転送(ブロック)
$8B	DMAMOV_A	DMA転送(アレイ・チェーン)
$8C	DMAMOV_L	DMA転送(リンク・チェーン)
$8D	DMAMOD	DMA状態チェック

参　考 (DO.L)	名　　称	機　　　　能
$8E	BOOTINF	ブート情報を返す
$8F	ROMVER	ROM バージョンを返す
$90	G_CLR_ON	グラフィック画面の初期化と表示モードの設定
$94	GPALET	グラフィックのパレット設定
$A0	SFTJIS	シフト JIS→JIS 変換
$A1	JISSFT	JIS→シフト JIS 変換
$A2	AKCONV	ANK→シフト JIS 変換
$A3	RMACNV	ローマ字→ANK カナ変換
$A4	DAKJOB	濁点処理(全角文字列)
$A5	HANJOB	半濁点(全角文字列)
$AE	OS_CURON	カーソル表示(IOCS$20～$2F に有効)
SAF	OS_CUROF	カーソル非表示(IOCS$20～$2F に有効)
SB0	RESERVED	———
$B1	APAGE	グラフィック画面の書き込みページ指定
$B2	VPAGE	グラフィック画面の表示ページ指定
$B3	HOME	グラフィック画面の表示位置設定
$B4	WINDOW	グラフィック画面のウィンドウ設定
$B5	WIPE	グラフィック画面の画面クリア
$B6	PSET	グラフィック画面のポイント・セット
$B7	POINT	グラフィック画面のポイント・ゲット
$B8	LINE	グラフィック画面のライン
$B9	BOX	グラフィック画面のボックス
$BA	FILL	グラフィック画面のボックス・フィル
$BB	CIRCLE	グラフィック画面のサークル
$BC	PAINT	グラフィック画面のペイント
$BD	SYMBOL	グラフィック画面のシンボル (クリッピングなし)
$BE	GETGRM	グラフィック画面の画面ゲット
$BF	PUTGRM	グラフィック画面の画面プット
$C0	SP_INIT	スプライト画面の初期化
$C1	SP_ON	スプライト画面の表示
$C2	SP_OFF	スプライト画面の非表示
$C3	SP_CGCLR	PCG クリア
$C4	SP_DEFCG	PCG 定義
$C5	SP_GTPCG	PCG 読み込み
$C6	SP_REGST	スプライト・レジスタの設定
$C7	SP_REGGT	スプライト・レジスタのチェック
$C8	BGSCRLST	BG スクロール・レジスタ設定
$C9	BGSCRLGT	BG スクロール・レジスタ・チェック
$CA	BGCTRLST	BG コントロール・レジスタ設定
$CB	BGCTRLGT	BG コントロール・レジスタ・チェック
$CC	BGTEXTCL	BG テキスト・クリア
$CD	BGTEXTST	BG テキスト設定
$CE	BGTEXTGT	BG テキスト・チェック
$CF	SPALET	スプライト・パレット設定 チェック

コールをマスターしたあとにはIOCS コールもレパートリに含めておきたいものです．
IOCS コールそのものは，

```
trap    #15
```

だけで行なえますが，その前に**コール番号**を D_0にセットしたり，他のパラメータをレジスタに与えるという準備が必要です．たとえば B_PRINT では，

```
moveq.l     #$21,d0
```

命令などをコール番号セットに使います．しかし，このままではリストを見ても＄21の意味がわかりにくいので，C コンパイラ・セットにある **iocscall.mac** をインクルードし，

```
moveq.l     #B_PRINT,d0
```

のように記述する方法を使うのがよいでしょう．

3-6 自動パワーON，パワーdownの実験プログラム

X68000には自動パワーOFF機能があり，アセンブラを使えば比較的簡単に作動できます．

しかし，この機能は安全のためにパワースイッチがOFFにならないと働かないようになっています．パワー・スイッチを押すと，本来のパワーOFF機能が働くので，そのままではせっかく作ったプログラムも「無用の長物」となってしまいます．

そこで，アラーム機能を使ってOFF状態から自動的に立ち上げるようにすれば，スイッチはOFF状態のままなので，パワーダウン・プログラムにも活躍の場ができます．

ここでは以上で説明したことを実験するため，2つのプログラムを作りました．

1つはタイマ・セットを行なうもので，図3.13に示すようにIOCSコールを使っています．このプログラムは，Time値で，曜日(日曜＝0起算)，日，時(24時間時)，分を与えてタイマ・セットを行ないます．次に

Push　power-switch　off

のメッセージを出して，パワー・スイッチがOFFになる(操作者がOFFにする)のを待ちます．Time値を変えてアセンブルすれば，任意の時刻にタイマがセットできるので，実験したい読者は試してみるとよいでしょう．なおD_2にセットする値は，自動立ち上げ後，OFFまでの時間で，分単位で与えます．ここでは600分(10時間)にしています．

もう1つのプログラムは，タイマによって立ち上がったあとで，プログラムによって自動パワーdownを行なうものです(図3.14)．

パワーdown後，セットした時刻になると，自動的にスイッチが入り，Human68Kが起動されるので，ここで自動パワーdownプログラムを実行させます．
スクリーンには，

Push　any　key

●図3．13　テスト用アラーム・セット・プログラム

```
     1 00000000                          *
     2 00000000                          *     Alarm Set
     3 00000000                          *
     4 00000000                                include ¥include¥iocscall.mac
     4 00000000                               `.list
     5 00000000
     6 00000000 =01031425                Time      equ      $01_03_14_25
     7 00000000 =0000000F                IOCS_call equ      15
     8 00000000
     9 00000000                                .text
    10 00000000                          powent:
    11 00000000 705E                           moveq.l  #_ALARMSET,d0
    12 00000002 223C01031425                   move.l   #Time,d1
    13 00000008 243C00000258                   move.l   #600,d2
    14 0000000E 93C9                           clr.l    a1
    15 00000010 4E4F                           trap     #IOCS_call
    16 00000012
    17 00000012 43F9(02)00000000               lea      msg1,a1
    18 00000018 7021                           move.l   #_B_PRINT,d0
    19 0000001A 4E4F                           trap     #IOCS_call
    20 0000001C
    21 0000001C                          loop:
    22 0000001C 60FE_0000001C                - bra.s    loop
    23 0000001E
    24 0000001E                                 data
    25 00000000 5075736820706F77         msg1  dc.b     "Push power-switch off",$0d,$0a,
$0
            65722D7377697463
            68206F66660D0A00
    26 00000018
    27 00000018                                end      powent
```

が表示されるので，1分ほど待って適当なキーを押すと，その瞬間パワーdown動作を開始します．このとき，自動起動後1分以内では再度アラームが働いてしまうので，注意が必要です．なぜなら，分カウントがアラーム時刻のままだからです．

プログラムでは，キー入力にIOCSコールのB_KEYINP(1文入力)を使い，入力されていない間はループさせるようにしています．

また，パワーdownシーケンスでスーパーバイザ領域をアクセスしなければならないので，このため事前にDOSコールの_SUPERを用いてスーパーバイザ・モードに切り換えます．

パワーdownシーケンスは，第1部で説明したとおりのことを行なえばよく，パワースイッチ(Power off control)用のアドレスに$0，$F，$Fを連続して書き込みます．このプログラムでは，その直後特権命令のhaltを使ってCPUを停止させています．

●図3．14 テスト用パワーdownプログラム

```
     1 00000000                        *
     2 00000000                        *      Auto Power down Test program
     3 00000000                        *
     4 00000000                                include ¥include¥iocscall.mac
     4 00000000                                .list
     5 00000000                                include ¥include¥doscall.mac
     5 00000000                                .list
     6 00000000
     7 00000000 =0000000F              IOCS_call equ   15
     8 00000000 =00E8E00F              powsw     equ   $e8e00f         Hardware Power s
witch Address
     9 00000000
    10 00000000                                .text
    11 00000000                        wait:
    12 00000000 43F9(02)00000000               lea      msg2,a1
    13 00000006 7021                           move.l   #_B_PRINT,d0    Display Message
    14 00000008 4E4F                           trap     #IOCS_call
    15 0000000A                        loop:
    16 0000000A 7000                           moveq.l  #_B_KEYINP,d0   Key-input 1 char
acter
    17 0000000C 4E4F                           trap     #IOCS_call
    18 0000000E 4A80                           tst.l    d0              If Null
    19 00000010 67F8_0000000A                  beq.s    loop               then loop
    20 00000012
    21 00000012 42A7                           clr.l    -(sp)
    22 00000014 FF20                           dc.w     _SUPER          Change to Superv
isor Mode
    23 00000016 588F                           addq.l   #4,sp
    24 00000018
    25 00000018                        *
    26 00000018                        *      Power down Sequence
    27 00000018                        *
    28 00000018                        powdn:
    29 00000018 13FC000000E8E00F               move.b   #$00,powsw
    30 00000020 13FC000F00E8E00F               move.b   #$0f,powsw
    31 00000028 13FC000F00E8E00F               move.b   #$0f,powsw
    32 00000030 4E722700                       stop     #$2700
    33 00000034
    34 00000034                                .data
    35 00000000 5075736820616E79       msg2    dc.b     "Push any key",$0d,$0a,$0
            206B65790D0A00
    36 0000000F
    37 0000000F                                end      wait
```

第4章

デバッガの使い方

4-1 デバッガの立ち上げとコマンド形式

プログラムの動作に異常があるとき，機械語命令レベルで診断できるのがデバッガです．たいがいのプログラムは作成直後に何らかの「虫」をもっているのが普通なので，アセンブラでは**デバッガ**を使用するケースが少なくありません．

デバッガの立ち上げは，マニュアルでは，

db ［<スイッチ>］［<ファイル名>］［<コマンド・ライン>］

のようになっています．しかし，スイッチはほとんどの場合使用せず，ファイル名はターゲット・プログラムが収容されているものを指定し，コマンド・ラインはそのプログラムへのコマンドを記述するので，次のように覚えておいてさしつかえありません．

db <ターゲット・プログラムの立ち上げ記述>

これによってターゲット・プログラムが読み出され，メモリにロードされます．ただし，この段階でただちに実行が開始されることはなく，pc(プログラム・カウンタ)に実行開始点のアドレスがセットされた状態で停止しています．このときスクリーンには各レジスタの初期値が表示され，デバッガ・コマンドの入力待ちを意味するプロンプト(—)が出ています．

ここで使用されるデバッガ・コマンドは，表4.1のようになります．また，コマンドは，

<コマンド>　　［<パラメータ>］

の形式で与えられ，パラメータのうちアドレス値などの変数は，“.”を先頭にしてレジスタ名などを指定し，その現在値を利用できます．パラメータは，変数を演算子によって組み合わせ，数式でも与えられま

アセンブラで作成したプログラムを走らせると，暴走するとか，原因不明の異常が発生するなどのトラブルは日常茶飯事に起きます．

このようなときデバッガの登場となるのですが，16ビット機のデバッガでは逆アセンブラが使えるのが普通で，Human68K の場合もその例にもれません．

デバッガを上手に使うコツは，怪しいと思われる部分の前後にブレーク・ポイントをセットし，レジスタ値などの値を確認することです．その値が期待値と一致しているかどうかがプログラム・ミスと断定する際の決め手になるのですが，初心者にとっては，これが命令の実行結果を予測するためのよいトレーニングにもなります．

「虫」が出たからといってガッカリせずに，「なぞ解き」のゲームくらいに考えて挑戦して欲しいものです．その際の「武器」は，もちろんデバッガです．

●表4．1　デバッガ・コマンド一覧表

分　類	コマンド形式	機　能
ブレーク・ ポイント操作	B(bp) (address) B BC(<bp<) BD(<bp>) BE(<bp>) BR	ブレーク・ポイントの設定 ブレーク・ポイントの表示 ブレーク・ポイントの削除 ブレーク・ポイントの一時無効化 ブレーク・ポイントの復活 ブレーク・ポイントの通過回数のクリア
ターゲット・ プログラムの 実行	C string G(＝address) (address) T(＝address) (count) U(＝address) (count) HI	文字列のコマンド・ラインへの設定 プログラムの実行 トレース 表示なしトレース トレース・ヒストリ
プログラム内容の 表示と修正	L(<range>) A P Ps	逆アセンブラ機能 ライン・アセンブラ プログラムのアドレス表示 シンボル・テーブルの表示
メモリ内容の 表示と変更	D(size) (<range>) E(size) (address) F(size)<range>data M<range>address S(size)<range>data	メモリのダンプ メモリのエディット メモリの埋め込み メモリの集団コピー メモリの探索
レジスタ，システム 変数の表示と変更	X X(reg) Z Z(num＝exp)	レジスタの表示 レジスタの更新 システム変数の表示 システム変数の設定
ファイルの利用	W filename,<range> R filename	ファイルへの書き出し ファイルの読み込み
その他のコマンド	?(exp) ??(exp) ¥ Y/N H V Q	計算結果の表示(16進) 計算結果の表示(10進) コマンド・ラインの繰り返し オペレータへの問い合わせ ヘルプ機能 コンソール切り換え デバッガの終了

す．

なお，デバッガのスイッチには，

／R

があり，このスイッチはRS-232Cポートを利用して外部のパソコンなどから操作するときに使います．

デバッガには逆アセンブラも用意されていますが，デバッグする場所を正確に把握するため，アセンブル時のリストを用意しておくのがベストです．

4-2　ブレーク・ポイントの設定，解除，一時無効化，復活

プログラムの要所々々に中断点(**ブレーク・ポイント**)を設定し，個々のポイントにおけるレジスタなどの現在値を調べることは，デバッグの基本でもあります．

ブレーク・ポイントは，結果を参照したい命令の次に並ぶ命令のアドレスを指定します．

デバッガはＧコマンドで実行を開始(または再開)するとき，ブレーク・ポイントとして指定されているアドレスの命令コードを退避し，かりに未実装命令に書き換えます．こうしておくと，プログラムがその場所の命令を実行しようとしたとき割り込みが発生し，デバッガに戻ることができるのです．このときデバッガはブレーク・ポイントの内容をすべて復元するので，ダンプや逆アセンブルを行なっても元の内容が参照できます．

ブレーク・ポイントを設定するコマンドは，

B [〈番号〉]　〈アドレス〉　[〈回数〉]

により与えます．番号は他のブレーク・ポイントを操作するコマンドで個々のポイントを区別するためのもので，０～９までの値で設定します．省略すると，空き番号が順次割り当てられます．番号を指定するときは，コマンドの“B”に続けて(スペースを空けずに)記述しないと，次のパラメータであるアドレスと区別がつかなくなります．回数は，そのブレーク・ポイントを通過し，何回目で停止するかの指定です．省略すると１がとられ，最初の通過と同時に停止することになります．

ブレーク・ポイントの解除は，

BC {〈番号〉}

コマンドによります．このときの番号はＢコマンドで設定した値，または自動的に割り当てられた値を指定し，複数個並べて同時に解除できます．

番号がわからなくなったら，

B

だけのコマンドを入力すると，現在のブレーク・ポイントのリストを参照できます．リストは番号，有効(e)／無効(d)の区別，アドレス，回数指定値，現在までの通過回数(各16進値)の順で表示されます．

ブレーク・ポイントを一時的に無効にするには，

BD {〈番号〉}

コマンドを使います．また，無効にしたものを復活させるには，

BE {〈番号〉}

コマンドによります．BD，BEコマンドですべてのブレーク・ポイントを対象にするときは，番号の代わりに“＊”を指定します．

なお，都合でブレーク・ポイントの通過回数をゼロに戻したいときは，

BR

コマンドによって，全部の回数をまとめて初期化できます．
ブレーク・ポイントは，Gコマンドによっても設定できます．

4-3 ターゲット・プログラムの実行

ターゲット・プログラムの中には，実行時にパラメータを必要とするものがあります．もしデバッガ起動後にパラメータを渡す必要が生じたときは，実行前に

C 〈文字列〉

コマンドによって与えます．
ブレーク・ポイントなどが設定されて，実行を開始するときは，

G

だけでそのプログラムの実行開始点からスタートします．ブレーク・ポイントで停止したあとの再開時にも，アドレス値の入力は必要としません．都合により特定のアドレスから実行する必要のあるときだけ

G= 〈アドレス〉

形式で記述すればよいのです．いずれの場合も，間にスペースを置いて，あとで**ブレーク・ポイント・アドレス**を指定できます．一般にブレーク・ポイント記述は1個だけにするのが望ましく，複数個(最大10個まで)指定しても，そのうち最初に通過したものだけしか効果がありません．なぜなら，Gコマンドのブレーク・ポイントは，通過後に自動的に解除されます．つまり，1回しか効かないからです．このため，条件ブランチのあとなど，行く可能性のあるところに1つずつ複数個指定することには意味があります．
68000はCPU自身がトレース・モードをもっていますが，これを利用したのが，

T [= 〈アドレス〉]　[〈カウント〉]

コマンドです．ここで，アドレスの考え方はGコマンドとまったく同じで，カウントは停止するまでの実行命令数(省略値は1)です．この方法では命令を少しずつ実行しながら調べることになるため，ブレーク・ポイントは必要としません．コマンド実行中は，1命令ごとにレジスタ値と，次の命令の逆アセンブル結果を表示します．
毎回表示されるのがわずらわしいときは，

U [= 〈アドレス〉]　[〈カウント〉]

コマンドを使うと，カウントで指定された命令数になるまで表示がバイパスできます．言い換えると，このコマンド1件について1回しか表示されません．カウントを省略した場合(省略値：1回)，結果はTコマンドと同じです．
命令の実行経路をトレースしたいときは，Uコマンドで任意のステップ数だけ進めておいて，

HI

コマンドを実行します．その結果，実行した順序でPC値が表示されます．ただし，このためのバッファは8ステップ分しかなく，オーバーした場合は古いものから捨てられます．また，今回のトレースが8ステップに満たない場合は，以前の残りが前方に付き，合計8ステップ分となって表示されます．その際，以

前にトレースをしていないときは，その部分のPC値はゼロとして残っています．

4-4 プログラム内容の表示と修正

プログラムの命令部分を参照するには，ダンプによるよりも逆アセンブルするほうが効果的です．逆アセンブルを行なうためのコマンドは，

L [<開始アドレス>　[<終了アドレス>]]

です．立ち上げ直後，アドレス全体の省略時はプログラムの実行開始点から逆アセンブラされます．これ以降の省略は，前回逆アセンブルした次のポイントが採用されます．終了アドレス省略時(全体省略時も含む)は，8行だけリスティングします．

命令単位にアセンブルしながら訂正する場合は，

A [N] [<アドレス>]

コマンドを使います．アドレス省略時の扱いは，Lコマンドと同様です．

このコマンドを立ち上げると，ライン・アセンブル・モードとなって，行(命令)単位に逆アセンブルされ(ただし“N”が指定されていると逆アセンブルは行なわれない)，ニーモニック1行分(1命令)の入力が要求されます．ここで入力した結果は機械語に変換され，以前の内容を置き換えます．なお，ここでは，“_EXIT”などのDOSコール・シンボルがそのまま記述できます．これらのシンボルは，小文字で書いてもかまいません．ライン・アセンブル・モードを終了したいときは，

. ↵　(ピリオド)

または

[CTRL]+[C]

を入力します．

上記の各コマンドは，アドレスを省略することで実行開始点から動作させることができるので，とくにアドレス範囲の情報はいらないことが多いのですが，プログラムによってはメモリのどの範囲にロードされているのか知りたい場合があります．そのようなときは，

P

コマンドより，デバッガ先頭アドレス，ターゲット・プログラム先頭アドレス，同終了アドレス，同実行開始アドレスやシンボル・テーブルがあるときはシンボル・テーブル・アドレスを表示できます．

シンボル・テーブルはリンカによってglobal(グローバル・シンボルの宣言)，xdef(外部定義名の宣言)を含むモジュールが処理された場合に作成されるもので，リンカの／Xスイッチを使わない限りデバッガで参照できます．参照の際に使用するコマンドは，

PS [<シンボル>]

です．ここで，シンボルを省略すると全シンボルが表示されます．また，シンボルとして1文字だけ入力すると，その文字を先頭とするすべてのシンボルが表示されます．その他は指定したシンボルを検索し，その値とシンボル名を表示します．

これらのシンボル名は，DOSコール・シンボルと同様，デバッガのコマンドにおいて式の中で利用できます．

4-5 メモリ内容の表示と変更

メモリの内容を調べる際，普通はダンプを行ないます．そのとき，コマンドは

D [<サイズ>]　[<開始アドレス>　[<終了アドレス>]]

形式で与え，サイズは省略時ワード単位となります．ここでいう「単位」とは，16進表示の区切りのことで，普通はワード単位が最も見やすいのが実態です．

範囲指定の開始アドレス，終了アドレスについては，省略した場合，次の値がとられます．すなわち，開始アドレスは前回ダンプした範囲の次のポイントになります．まだダンプしていないときは，ターゲット・プログラムの先頭のアドレスが入ります．終了アドレスは，開始アドレスに127を加えた値となり，128バイト分のダンプが得られることになります．

メモリへのデータの書き込みは，個別書き込みにはＥコマンド，同一データの連続大量書き込みにはＦコマンドが使われます．

Ｅコマンドは，

E [N] [<サイズ>]　[<アドレス>]　[<データ>]

のフォーマットで入力します．連続してデータを書き込む場合は，コマンド上のデータを省略するとエデット・モードとなり，アドレスが自動的に加算され表示されるので，それに従ってデータを入力します．このモードから抜けるには CTRL ＋ C を押します．バイト単位のときサイズは“P”も指定可能で，このときエデット・モードの自動加算値は２が採用されます．この機能は，movep 命令の対象となる周辺装置を考慮したものです．コマンドは“E”の次に“N”を付けると，エディット・モードで変更前のデータの表示がバイパスされます．しかし，一般には確認のため表示させたほうがよいと思います．

Ｆコマンドは，次のフォーマットで入力されます．

F [<サイズ>]　<開始アドレス>　<終了アドレス>　<データ>

ここで，サイズの省略値はワードとなります．

メモリ内容のグループ転送(コピー)をするときは，

M <開始アドレス>　<終了アドレス>　<コピー先アドレス>

コマンドによります．このとき，転送元，転送先の領域に一部重複があるときは，期待したとおりの結果が得られないことになりがちなので注意が必要です．

メモリ内容の変更を行なう際には，変更すべき場所を探すことから始めなければならないことが少なくありません．その場合，

S [<サイズ>]　<開始アドレス>　<終了アドレス>　<データ>

コマンドが大いに役立ちます．Ｓコマンドは，開始アドレス～終了アドレスの範囲内で指定されたデータを検索し，該当するアドレスをすべて表示します．データは文字列も可能で，その場合サイズはバイトとして扱われます．また，数値の場合のサイズの省略値はワードが採用されます．

4-6 レジスタ，システム変数の表示と変更，およびその利用

デバッガではコマンドによってレジスタ値を参照したり，変更できます．同じことは，**システム変数**と呼ばれるデバッガ内の変数に対しても行なうことが可能で，レジスタ値のセーブや，よく使う定数値など

を置いておく作業用レジスタとして利用できます．システム変数は**Z0～9**の仮想レジスタで，デバッガのときだけ使われるものです．

一般のレジスタ値を参照するには，

X

コマンドを使います．これによって，PC，USP，SSP，SR，$D_{0\sim7}$，$A_{0\sim7}$(A_7はスーパーバイザ・モードのときSSP，ユーザ・モードのときUSP)の値と，続いてPC値が示すアドレスにある1命令の逆アセンブル結果が出力されます．SRの内容は，全体の16進値とX，N，Z，V，Cフラグの個別の値が分解されて表示されます．一般に，私たちが作るプログラムはユーザ・モードなので，上位のビット参照する必要はあまりありません．しかし，仮にビット13がONになっているときはスーパーバイザ・モードになっているので，故意でないとすれば何らかの異常が考えられます．したがって，このビットだけは注意しておきたいものです．

レジスタ値の変更は，

X 〈レジスタ名〉

コマンドにより現在値が表示されるので，それに続いて設定したい値を入力します．レジスタ名の指定は，上記のすべて(PC～A_7)について可能です．

一方，システム変数の表示は，

Z

コマンドによって表示できます．変数の設定には，

Z 〈番号〉 ＝ 〈式〉

コマンドを使います．レジスタ値の変更と異なり，“＝”が入ったり，コマンドで直接値を入力する点に注意が必要です．

レジスタやシステム変数の値は，先頭に“.”を付けて参照，代入できます．たとえば，デバッガ立ち上げ直後に，

Z0＝.PC

とやっておけば，デバッガ中いつでもプログラムの実行開始アドレスを“.Z0”として参照できます．具体的には，この後，

L　.Z0

で実行開始点からの逆アセンブルができます．

また，同様に

Z1 ＝ 〈データ領域の先頭アドレス〉

を実行しておけば，常に，

D　.Z1

でデータ・エリア先頭からのダンプができます．

4-7 デバッガへのファイルの利用

デバッガで発見された「虫」は，ライン・アセンブラなどでパッチ*すれば，メモリ上で正しい状態にできます．しかし，この状態はデバッガを立ち上げている間だけしか持続できず，Qコマンドの実行とともに消え去ってしまいます．もちろんデバッガの目的は，発見したプログラム・ミスをメモリ上で訂正することにあるので，一般にはそれで困ることはありません．

しかし，長時間にわたるデバッグなどでは，途中で電源OFFにしたいとか，パソコンに別な仕事をさせたいということも少なくなく，このような場合，中間結果を保存しておきたいという要求が生じます．

このためメモリ内容をセーブするコマンドとして，

W　<ファイル名>　<開始アドレス>　<終了アドレス>

が用意されています．反対に，セーブした内容をロードするには，

R　<ファイル名>［ <開始アドレス>］

コマンドを使います．Rコマンドでロード開始アドレスを省略すると，セーブしたときの開始アドレスではなく，**現在のPC値**が使われるので注意が必要です．なお，このとき作成されたファイルは単なるメモリのコピーにすぎず，デバッガの助けがないと実行できません．

一方で，実行形式のプログラムだけがあって，ソースがない場合，以上の機能だけでは，修正後の機械語プログラムを働かせたいときに常にデバッガが必要になって，操作上非常に不便です．そこで，物理ア

●図4．1　レコード番号の配置

サイド	セクタ	トラック			
		0	1	2	3 …………
0	1	00	16	32	48
	2	01	17	33	⋮
	3	02	18	34	
	4	03	19	35	
	5	04	20	36	
	6	05	21	37	
	7	06	22	38	
	8	07	23	39	
1	1	08	24	40	
	2	09	25	41	
	3	10	26	42	
	4	11	27	43	
	5	12	28	44	
	6	13	29	45	
	7	14	30	46	
	8	15	31	47	

＊機械語プログラムなどを直接修正すること．

ドレスを指定してファイルのロード，セーブができる機能も用意されています．このときの使い方は，先に目的のプログラムをロードして，パッチしたものを同じ物理アドレスに戻すのが無難です．ただし，一歩間違えると目的のプログラムだけでなく他のファイルまで破壊する危険な方法だけに，前もってディスクのバック・アップをとるなど万全の準備をした上でかかる慎重さが求められます．このときのロードには，

R@〈開始アドレス〉〈ドライブ番号〉〈レコード番号〉〈レコード数〉

コマンド，セーブには，

W@〈開始アドレス〉〈ドライブ番号〉〈レコード番号〉〈レコード数〉

コマンドが使われます．

ここで，ドライブ番号はAが0，Bが1番に対応します．1レコードは1,024バイトで，レコード番号は図4.1のように対応しています．

4-8　その他のデバッガ・コマンド

デバッガを使用中に，電卓代わりにして計算したいことが時々あります．計算が面倒なときは数値の代わりに式を使ってもよいのですが，これは同じ計算値を何度も使用するときは一度結果を出して知っておいたほうが操作が楽になるからです．デバッガの「電卓」は，16進で答えを出すことができたり，16進値を10進値に変換できたりするので，本物よりデバッグに便利です．電卓機能の結果を**16進値**で得たいときは，

？〈式〉

10進値で得たいときは，

？？〈式〉

コマンドを使用し，式の中には16進，10進値をごちゃまぜにして記述できます．もちろん，変数も使用できます．

コマンドは，“：”で区切って，1行に複数個実行順に並べることができます．その後に

￥

コマンドを記述すると，そのコマンド実行時に同じコマンド行の始めに戻すことができます．いわば**ループ**するわけです．しかしこのままでは永久ループになってしまうので，

Y／N

コマンドを入れて，これによって**続行**するかどうかの**問い合わせ**をします．

ここで“Y”を入力するとループが継続します．“N”は打ち切りたいときに入力します．

デバッガの機能の概要が理解できたら，コマンドを忘れてもマニュアルなどを参照する必要はありません．

H

コマンドで**ヘルプ機能**が働き，コマンドとその形式を表示してくれます．途中で**RS-232C**ポートに接続されている端末からの操作に**切り換え**たいときは，

V

コマンドを使います．反対に，RS-232C ポートから本体側に戻したいときも，同じコマンドを利用します．

デバッガのすべての操作を終わって終了させるときは，

Q

コマンドを入力します．このコマンドにより，デバッガ使用前の状態，すなわち Human68K の画面に戻ります．

付録　OS-9/68000への招待

1 究極のX68000の楽しみ方

Human68KはTSS(Time Sharing System)ができないので，68000CPUが本来もっている多重処理のたもの機能がフルに発揮されていません．

一方で，最初6809CPUのためのOSとしてマイクロウェアによって開発されたOS-9は，8ビットCPUでTSSができるということで一躍話題となりました．その後68000CPUの登場により，OS-9／68000へと舞台が移ってきましたが，68000は6809の思想を発展的に継承しているため，かなり増強されたものとなっています．

とりわけ8ビット版でネックになっていたメモリ容量の問題は，68000マシンの広大な領域のおかげでそれほど気にしなくてもよくなり，TSSも額面どおりエンジョイできるようになりました．

X68000のOS-9は，さらにパーソナル・ウィンドウ，マルチ・スクリーンをサポートし，μMACSスクリーン・エディタ，AV-BASIC(FM音源やスクリーンの制御を強化したバージョン)，Cコンパイラ，アセンブラ，リンカ，デバッガ，OS-9／NETなどのソフトが走ります．

Human68Kのソフトは，C言語によるものを除き他社機とほとんど互換性がありませんが，OS-9についてはかなりの程度互換性が期待できるのも大きな魅力です．標準的なデバイスを使用するプログラムならば，アセンブラで書かれたプログラムでさえもそのまま走ることが多いのです．

このことは，68000のポジション・インディペンデント(プログラムのロード位置が自由で，どの位置にあっても実行できること)な性格と，それに伴う移植性の良さの裏付けともなっていますが，こういった68000の設計思想を活かしたOSに接することが，究極のX68000の楽しみ方といえるかもしれません．

以下，OS-9の使い方，プログラミングの仕方などについて説明します．

2 スッキリしているファイル管理

OS-9のファイル管理では，Human68Kと同じように階層構造のディレクトリを採用しています．したがって基本的な考え方は説明の必要もないくらいですが，表現の仕方が異なります．

最も大きな違いは，カレント・ドライブまでがディレクトリの中に統合されていることで，その切り換えはチェンジ・ディレクトリ・コマンドで行なえます．

たとえばHuman68Kで，

A:￥B￥C

と表わされるファイル名は，

／D0／B／C

のように記述されます．D_0はディスク・ドライブ0を表わし，カレント・ディレクトリが／D_0にあるときは，同じファイルは

B／C

と表現できます．

OS-9は２種類のカレント・ディレクトリをもっており，１つは“**データ・ディレクトリ**”，もう１つは“**実行ディレクトリ**”と呼ばれています．一般にデータ・ディレクトリは，データ・ファイルやリロケータブル形式までのプログラム・ファイルに利用され，実行ディレクトリは実行形式のプログラム・ファイルを参照する(実行する，あるいはリンカなどで作成する)のに使われます．

Human68K と比較すると，カレント・データ・ディレクトリはカレント・ディレクトリに該当します．カレント実行ディレクトリは通常，

／D0／CMDS

に設定されており，このディレクトリには外部コマンドのためのプログラムやユーザ開発プログラムが収容されています．もし Human68K ならば，１つのカレント・ディレクトリで両者を兼ね，拡張のため PATH を使います．OS-9では他のディレクトリの内容を実行したければ，同様に環境変数 **PATH**(複数定義可)を前もって定義しておくか，カレント実行ディレクトリを目的の場所に変更します．

Human68K でバッチ・ファイルと呼んでいるものには，OS-9ではプロシージャ・ファイルが相当します．ただし OS-9には拡張子という概念がないので，“.bat”のように区別する記述がありません．言い換えると，ファイル名を見ただけでプロシージャ・ファイルかどうかを識別できないのですが，OS-9に慣れてしまうと Human68K のエディタなどで拡張子まで指定しなければならないのがかえってわずらわしく感じるくらいです．

3 コマンドも Human68K に似ている

Human68K のコマンドを理解しているユーザーにとって，OS-9のコマンドはすぐにでも使えます．

よく使うカレント・データ・ディレクトリの変更は，

chd　<パス>

で，コマンド名が違うだけです．この chd は，カレント・ドライブに相当するドライブ名の変更にも使えます．たとえば Human68K で“B：”の操作は，

chd ／D1

のようにします．

ディレクトリの新設，削除は，

```
makdir 〈パス〉……
deldir  〈パス〉……
```

で行ない，これも基本的にコマンド名が違うだけですが，パス名は複数個同時に指定できるようになっています．

ディレクトリをリスティングするときは，

```
dir 〈パス〉……
```

で，まったく同じです．

さらに

```
copy    〈コピー元パス〉〈コピー先パス〉
del     〈パス〉……
rename  〈旧ファイル名〉〈新ファイル名〉
```

なども同じように使えます．

ただし，これらのコマンドの細部(オプションなど)は「似て非なるもの」ですから，念のためにつけ加えます．

このように，私たちがよく使うコマンドは，Human68K とほとんど同じか，少し違う程度の差で操作できるのです．

Human68K と根本的に異なる点は，カレント実行ディレクトリの変更コマンド

```
chx 〈パス〉
```

が用意されていることです．これは chd と同じく内部コマンドで，実行前のカレント実行ディレクトリの位置に関係なく実行できます．

だいたいこういったコマンドを知っておくと，日常的な操作では支障をきたしません．

4 リダイレクト記号の意味

OS-9でも**標準入力，標準出力**という考え方は同じで，リダイレクト記号は，“＜(標準入力)”，“＞(標準出力)”というように，Human68K の記述と変わりません．

ただし，“＞＞”の意味が OS-9ではまったく異なり，標準出力へのアペンド(追加)はしません．この記号は**標準エラー出力**に割り当てられていて，プログラムからのエラー・メッセージ，通知などの出力に用いられます．理論的に独立しているので，標準出力と並行してリダイレクトする(“＞”と“＞＞”を同一コマンドに書く)ことも可能です．

OS-9のリダイレクト機能で強力な点は，コマンドでプロシージャ・ファイルの内容を実行するとき，

```
$  proc>filea
```

のように全体をリダイレクトできることです．このとき filea は起動時にオープンされ，プロシージャ・ファイルの処理が終了した時点でクローズされます．その間標準出力から送られたデータは，プログラムが異なっても切れ目なく filea にアペンドされていきます．

Human68K では，バッチ・ファイルから起動するときリダイレクト記号(＞)を書いてもエラーにはなりませんが，その代わり無視されます．「いけそうだ！」と思っていると，アテがはずれてガッカリということになります．

5 強力なコマンド・グループの実行

コマンド・グループの実行にはOS-9でも，Human68Kのマルチ処理に該当する**シーケンシャル実行**があり，セパレータには“；”が使われます．また，コマンド行の記述にカッコが使えるので，

$ (a;b)＞filea

によって，コマンドaとbの標準出力を連続してfileaに書き出すことができます．Human68Kではこのような機能がないので，逆にアペンドのためにリダイレクト記号(＞＞)が必要であるともいえるのです．

パイプライン実行はHuman68Kと同じ概念で，次に述べるコンカレント実行のコマンド同士の入出力を同期させたものと考えることができます．この場合，セパレータは“！”を使います．例をあげると

$ a!b

と書けば，コマンドaの標準出力がコマンドbの標準入力に接続されます．

同じ複数コマンド記述でも，セパレータとして&を用い，

$ a&b

と書くと，コマンドaとbは同時に独立して実行を開始します．これは**コンカレント実行**と呼ばれ，Human68Kにはなかった機能です．これを使えば，TSSが可能になります．

TSSを行なうときは，

$ a&

のようにして起動すると，shellコマンド・インタプリタと並行して走るため，コマンドaの実行中でもshellは別途コマンドを受け付け，実行させることができます．複数ユーザーのときは，

$ shell &

でshellを別に立ち上げることもできます．このとき，新たにshellをメモリにロードすることはなく，すでにロードずみのものがリエントラントに共用されるため，メモリが有効に使われます．

6 パーソナル・ウィンドウ

XOS-9を立ち上げると，**パーソナル・ウィンドウ**画面が現われます．

ウィンドウ各部の名称は付図1のように定義され，このような画面を複数個作ることができます．そのうち，現在キーボードが接続されているウィンドウを**カレント・ウィンドウ**といいます．カレント・ウィンドウの切り換えは，マウス・ポインタをそのウィンドウ内にもっていけばよいので非常に簡単です．

ウィンドウは，基本的には立ち上げる原因となったコマンドと運命をともにします．たとえば，マウス・ポインタが普通の位置にあるとき，右ボタンを押すと**ポップ・アップ・メニュー**が表示されます(付図2)．ここで右ボタンを押したままshellの位置までマウス・ポインタを移動してボタンを離すと，新しいウィンドウが作られると同時に新しいshellの動作が開始します．すなわちコマンド受け付け状態となり，このウィンドウ内で別途並行した処理が可能になるのです．

このウィンドウを消去するには，マウス・ポインタをコントロール・ボックス内に移動し，右ボタンを押します．そうするとコントロール・ボックス・メニューが表示され，右ボタンを押しながらマウス・ポインタを「ウィンドウの破壊」の位置に置いてボタンを離せば，shellが終了してウィンドウも除去されます．

もう1つの方法は，要するに「運命共同体」のshellを終了させればよいのですから，コマンド待ち状態で，

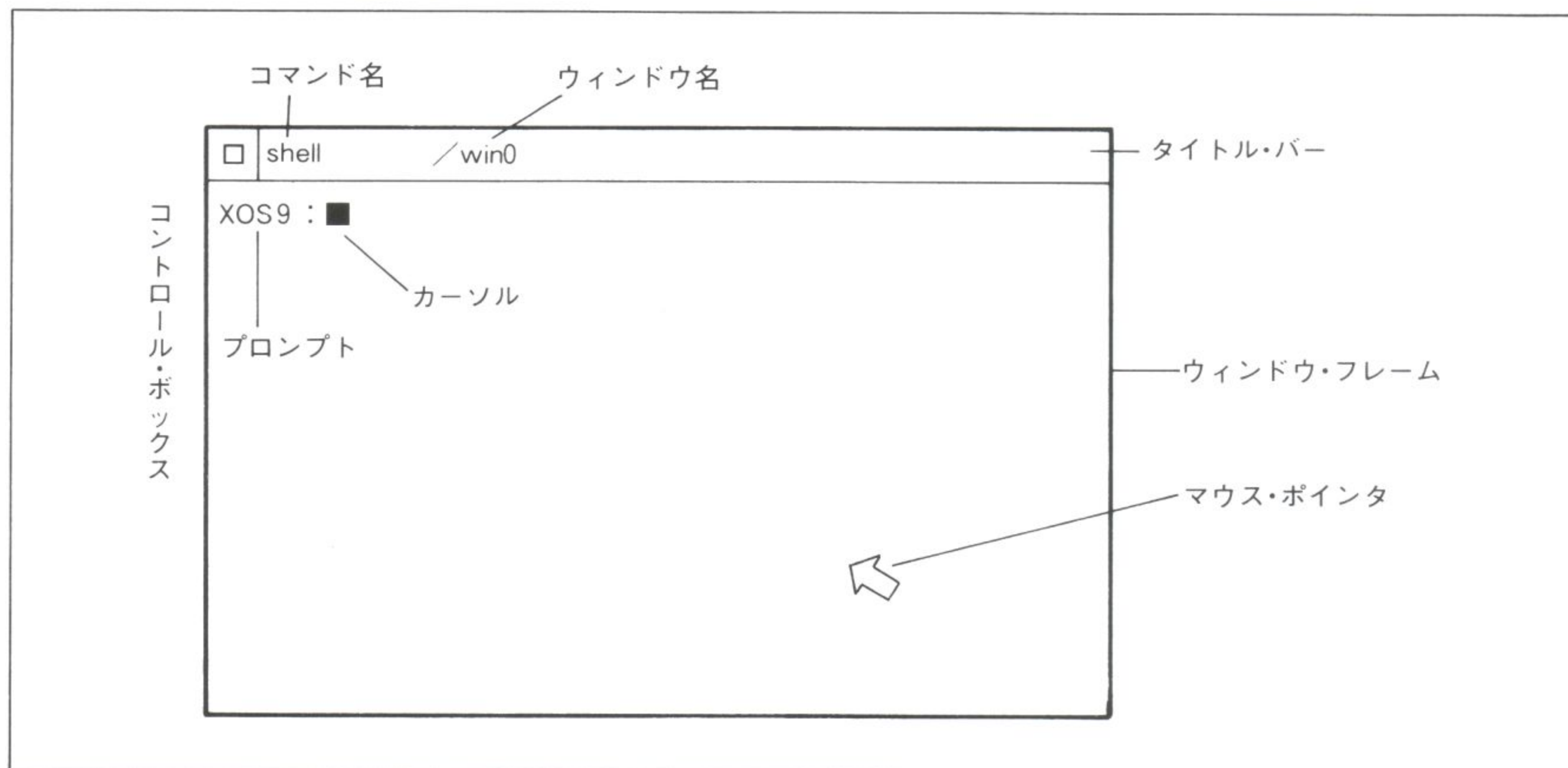
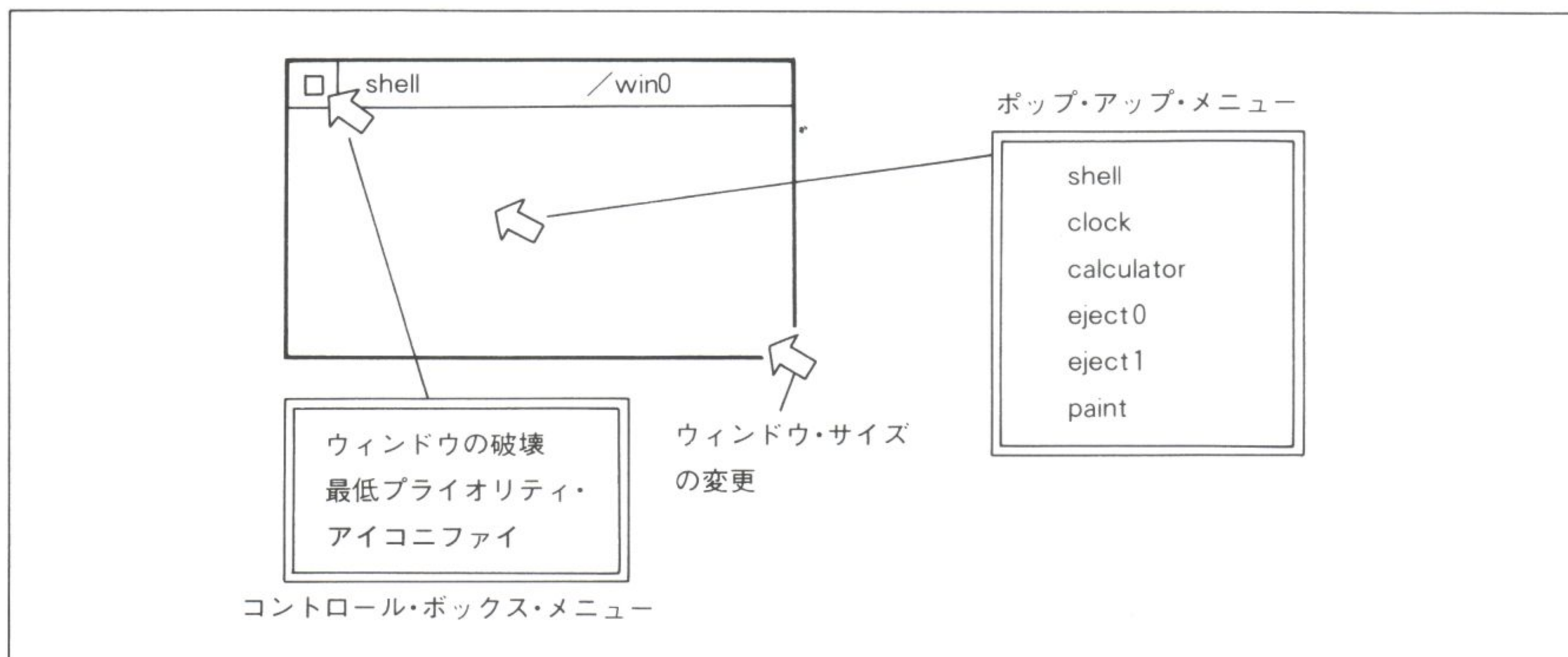
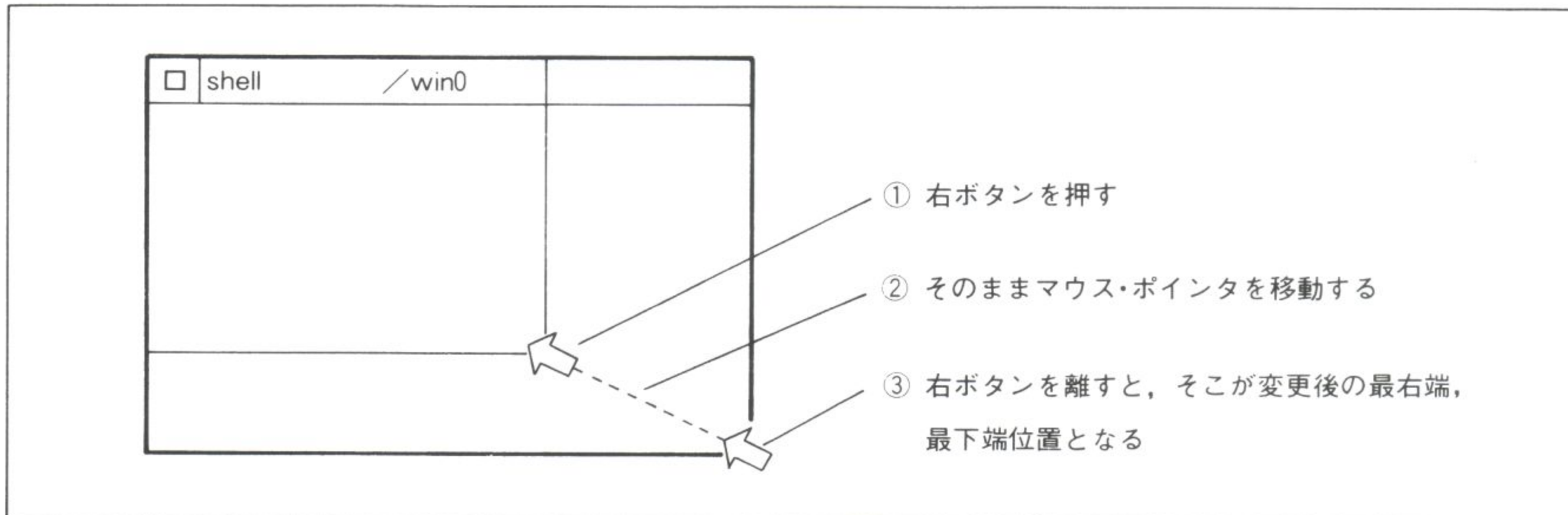
●付図1　ウィンドウ各部の名称

コマンド名
ウィンドウ名
shell　／win0
タイトル・バー
コントロール・ボックス
XOS9：■
カーソル
プロンプト
ウィンドウ・フレーム
マウス・ポインタ

●付図2　右ボタンを押す位置と動作の違い

shell　／win0
ポップ・アップ・メニュー
shell
clock
calculator
eject0
eject1
paint
ウィンドウの破壊
最低プライオリティ・
アイコニファイ
コントロール・ボックス・メニュー
ウィンドウ・サイズの変更

●付図3　ウィンドウ・サイズの変更

shell　／win0
① 右ボタンを押す
② そのままマウス・ポインタを移動する
③ 右ボタンを離すと，そこが変更後の最右端，最下端位置となる

●付表1　NEWWIN のオプション

オプション	機　　能
－?	ヘルプ機能(使用法の表示)
－c＝〈数値〉	ウィンドウ幅を文字単位で指定
－w＝〈数値〉	ウィンドウ幅をピクセル単位で指定
－r＝〈数値〉	ウィンドウの高さを文字単位で指定
－h＝〈数値〉	ウィンドウの高さをピクセル単位で指定
－u	ウィンドウ・サイズの固定(変更できなくする)

ESCキーを押すと，あっという間にそのウィンドウが消えてしまいます．

ウィンドウ・サイズの変更は，付図3のようにマウス・ポインタを端にもっていき，ここで右ボタンを押してそのまま希望する位置まで移動させ，そこでボタンを離します．変更できるのは終端だけで，始端は変えられません．

ウィンドウは新設するたびに，どんどん以前のものの前に作られていきます．したがって，その「下敷き」となるウィンドウは，スクリーン上で直接見えなくなってしまいます．そこで，表にあるウィンドウを裏に移動させるのがコントロール・ボックス・メニューの「最低プライオリティ」です．「アイコニファイ」というのはアイコン化するということで，邪魔なウィンドウをタイトル・バーの左端だけ残して透明にしてしまうものです．このときウィンドウは消されたわけではなく，マウス・ポインタをコントロール・ボックスに移して右ボタンをクリックすれば内容を復活させることができます．

コマンドでウィンドウを新設し，その上でプログラムを実行するには，

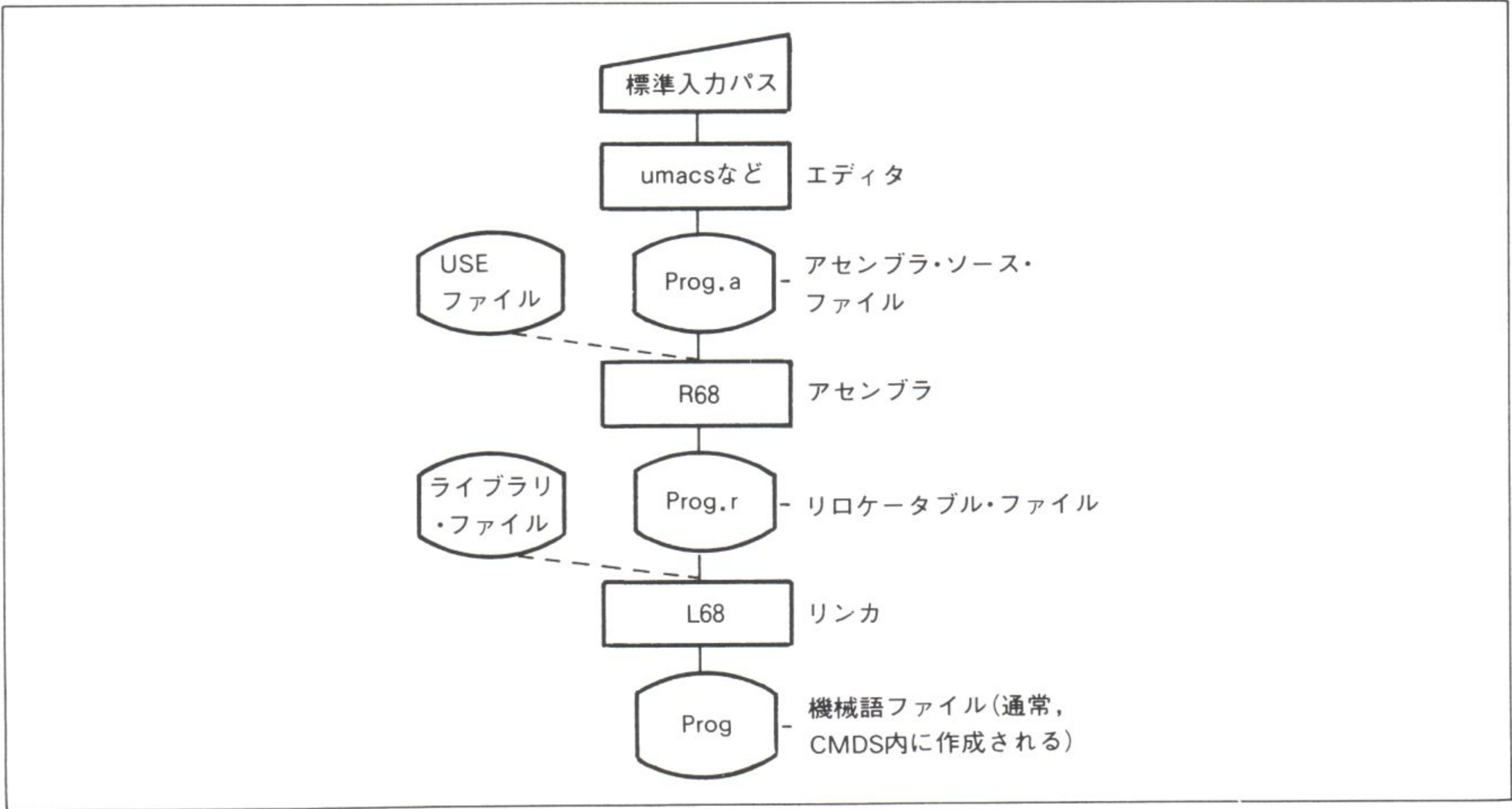
```
newwin  {<オプション>}  <プログラム名> [<パラメータ>]
```

を使います．オプションで指定できる内容は付表1のとおりです．

7 アセンブラによるプログラム開発

OS-9のアセンブラによるプログラム開発のプロセスを，付図4に示します．大筋ではHuman68Kと変わりませんが，プログラム名やファイル名の付け方が異なります．

●付図4　機械語ファイルが作成されるまでの一般的なプロセス

標準入力パス
umacsなど　エディタ
USE ファイル
Prog.a　アセンブラ・ソース・ファイル
R68　アセンブラ
ライブラリ・ファイル
Prog.r　リロケータブル・ファイル
L68　リンカ
Prog　機械語ファイル(通常，CMDS内に作成される)

●付表2　アセンブラのオプション一覧表

オプション	意　　味
－0＝〈ROF名〉	指定ファイルにROFを出力する
－1	リスト出力（さらに以下のオプション指定可）
－c	条件付き文も含めた出力
－dnn	ページ当たり行数（nn）の設定
－e	エラー出力の抑制
－f	ページ排出
－g	生成された全コードのリスティング
－n	リストから行番号を省略
－q	警告メッセージの抑制
－s	リストの最後にシンボル・テーブル出力
－x	マクロの展開結果も出力

●付表3　リンカのオプション一覧表

オプション	意　　味
－a	BSR命令で範囲を越えた場合，ジャンプ・テーブル参照に置き換える．
－e＝〈数　値〉	モジュール・エディション・ナンバーの指定．省略すると1になる．
－g	デバッガでシンボルを使用するとき参照するファイル（“.stb”付き）を出力する．
－j	ジャンプ・テーブルをマップにプリントする．
－l＝〈パ　ス〉	ライブラリのパスを指定する．このオプションは32個まで定義できる．
－m	psectのベース・アドレスをマップにプリントする．
－M＝〈数　値〉	スタック・サイズをKB単位に指定する．
－n＝〈モジュール名〉	モジュール名の指定．省略すると－oオプションのものが採用される．
－o＝〈パ　ス〉	実行形式プログラム収容ファイルのパス．
－p＝〈16進値〉	モジュール・ヘッダ内のアクセス・パーミッション値の指定．
－r	OS-9でないシステム用のモジュール作成．
－r＝〈16進値〉	同上でベース・アドレスを指定する．
－s	シンボルに割り当てられた相対アドレス値をマップに出力．
－w	－sオプションでアルファベット順に出力させる．
－z	標準入力からモジュール名を読み取る．
－z＝〈ファイル名〉	指定されたファイル名からモジュール名を読み取る．

OS-9ではファイル名に拡張子は付きませんが，普通

.a　…ソース・プログラム・ファイル
.r……リロケータブル・オブジェクト・ファイル(ROF)
.l……ライブラリ

のように，末尾に拡張子的な文字列を加えます．これはあくまでファイル名の一部であって，“.”もファイル名を構成する一文字にすぎず，セパレータでないことに注意が必要です．

したがって，アセンブラなどではファイル名の指定に際し“.”以下を省略するということはできません．逆の見方をすれば，ファイル名は必ずしも上記のルートに基づいて付けなくてもかまわないのですが，そうすると後述するmakeコマンドで利用されるmakefileの内容が面倒になるとか，プログラムの管理上わかりにくくなるなどの問題が発生します．ここではそういった前提で，暗黙のルールに基づいて説明を進めていきたいと思います．

さて，アセンブラを起動するときは，

r68　〈ソース・ファイル名〉{〈オプション〉}

コマンドを使います．オプションの内容は，付表2のとおりで，“－0＝”でROF名を指定するほかは，リストの出力に関するものばかりです．表の点線以下のオプションは，リスト出力時(－1オプションを指定)の細部の指定を行なうものです．

リンカの立ち上げは，

l68　〈メイン・モジュール名〉 [{〈ROF 名〉}] {〈オプション〉}

コマンドによります．メイン・モジュールも ROF の１つで，ROF 名が並ぶときは，先頭(左)に書かれたものがメイン・モジュールとみなされます．残りの ROF はメイン・モジュールと結合することが必要なものを列挙します．オプションは付表３のとおりで，システム・リクエストを使うときなどライブラリが必要なときは−l，実行形式の機械語ファイルを出力するときは−o により該当ファイル名を指定します．たとえば，次のように書きます．

−l＝／D0／LIB／SYS.L　　(システム・リクエスト使用時)
−o＝PROG　　(実行形式の PROG ファイルを作成するとき)

ここで，−o オプションは，CMDS ディレクトリに実行形式機械語ファイルを出力するとき，パス上位の記述が省略できます．もしほかのディレクトリに出力したいときは，全パス・リスト を記述します．

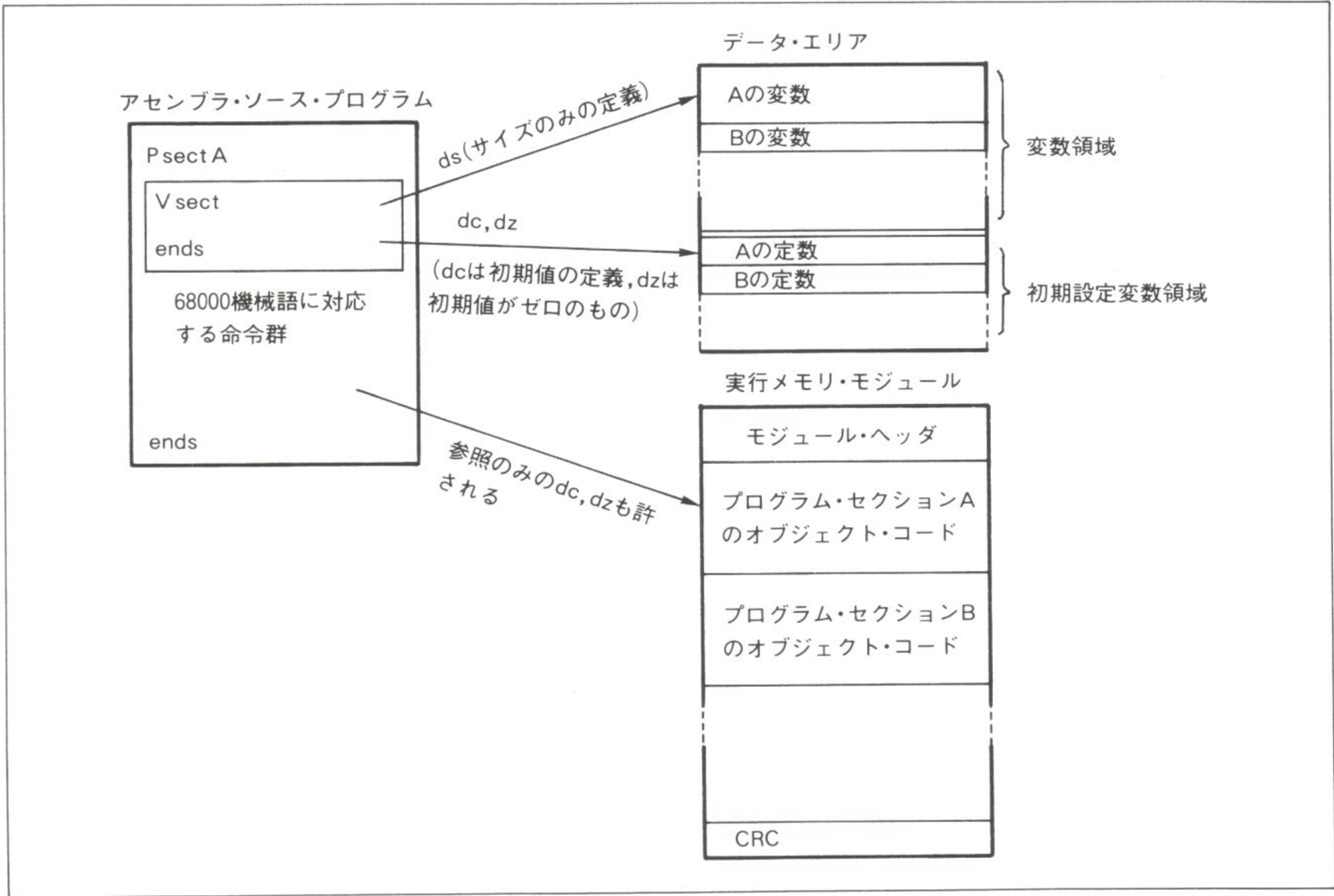
8　プログラム・セクションとデータ定義

Human68K と同様に，OS-9のアセンブラにも**プログラム・セクション**が存在します．OS-9の場合は，付図５のように，**psect** で定義された実行メモリ・モジュールが主として参照するデータ領域を，**vsect** で定義するのが一般的なパターンです．

データ領域内で使用できる擬似命令は，

dc | .b / .w / .l | **〈データ値〉**　(初期値のあるデータ)

●付図５　プログラム・セクションとメモリ上の実際のロケーションとの対応

アセンブラ・ソース・プログラム
PsectA
Vsect
ends
68000機械語に対応する命令群
ends
ds(サイズのみの定義)
dc,dz
(dcは初期値の定義，dzは初期値がゼロのもの)
参照のみのdc，dzも許される
データ・エリア
Aの変数
Bの変数
変数領域
Aの定数
Bの定数
初期設定変数領域
実行メモリ・モジュール
モジュール・ヘッダ
プログラム・セクションAのオブジェクト・コード
プログラム・セクションBのオブジェクト・コード
CRC

●付表 4　モジュール・タイプの記述内容

シンボル名	定義値	意　味
—	0	使用せず(システム・コールのワイルド・カード値)
Prgrm	1	プログラム・モジュール
Sbrtn	2	サブルーチン・モジュール
Multi	3	マルチ・モジュール(予約ずみ)
Data	4	データ・モジュール
—	5〜10	予約ずみ
Trap Lib	11	ユーザ・トラップ・ライブラリ
Systm	12	システム・モジュール
Flmgr	13	ファイル・マネージャ
Drivr	14	デバイス・ドライバ
Devic	15	デバイス・ディスクリプタ
—	16〜255	ユーザ定義

●付表 5　言語の記述内容

シンボル名	定義値	意　味
—	0	使用せず（システム・コールのワイルド・カード）
Odjct	1	68000 機械語
ICode	2	Basic の I コード
PCode	3	Pascal の P コード
CCode	4	C の I コード（予約ずみ）
Cbl Code	5	Cobol の I コード（予約ずみ）
Frtn Code	6	Fortran の I コード（予約ずみ）
—	7〜15	予約ずみ
—	16〜255	ユーザ定義

dz | .b / .w / .l | **<サイズ>**　　(初期値0のデータ)

ds | .b / .w / .l | **<サイズ>**　　(初期値なしのデータ)

です．このうち dc と dz については，vsect 内だけでなく実行モジュールの中に置いて参照できます．その場合，内容値を変更することはリエントラントの条件に反するので，絶対に避けるべきです．

プログラム・セクションの始まりは，通常次のように記述します．

psect　**<セクション名>，<タイプ／言語>，<属性／リビジョン>，<エディション>，<スタック・サイズ>，<エントリ・ポイント>**

ここで，モジュール・タイプ(上位 8 ビット)，言語種類(下位 8 ビット)は付表 4，付表 5 のとおりです．属性は ReEnt(リエントラント)のみで上位 8 ビットに与え，下位 8 ビットにはリビジョンを入れます．エディションは同一リビジョン内の内番です．スタック・サイズはこのモジュールが必要とするスタック領域の大きさ，エントリ・ポイントはメイン・プログラムについてのみ実行開始点のラベルを記述します．

上記の表にあるようなシンボル名は，defsfile を通して参照することによって，定義値と置き換えることができます．その場合プログラムの冒頭で，擬似命令の

use／d0／defs／defsfile

により該当内容を取り込んでおく必要があります．use は Human68K でいえば include と同じで，defsfile

はいわばインクルード・ファイルです．

vsect はデータ領域の先頭(ただしそのプログラム・モジュール内で)に書かれ，終わりには ends を置きます．psect も最後は ends でしめくくるようになっており，vsect 定義は psect の入れ子になるのが普通です．

9 アセンブラ・プログラムの作り方

OS-9のプログラム本体は，リエントラントであることが前提となっているので，データ領域は完全に別な場所に取得されます．すなわち，１本のプログラムが複数のユーザから同時に利用されても矛盾が生じないよう，データ領域の配分は実行段階にならないと決められません．また，複数ユーザによる同時実行の際は，このようなデータ領域がユーザごとにとられます．要するに，プログラム本体も，データ領域も，アドレスは実行直前まで決まらないのです．

このような背景でいったいどうやってプログラミングするのかといえば，答えは付図６に示されています．

つまり，A6にデータ領域の先頭アドレスが与えられるので，データ領域へのアクセスは A6相対形式(ディスプレースメント付きアドレス・レジスタ間接)で行なうようにするのです．一方，本体内のラベルを参照する際は，ディスプレースメント付き PC 相対アドレッシングを使います．たとえば，

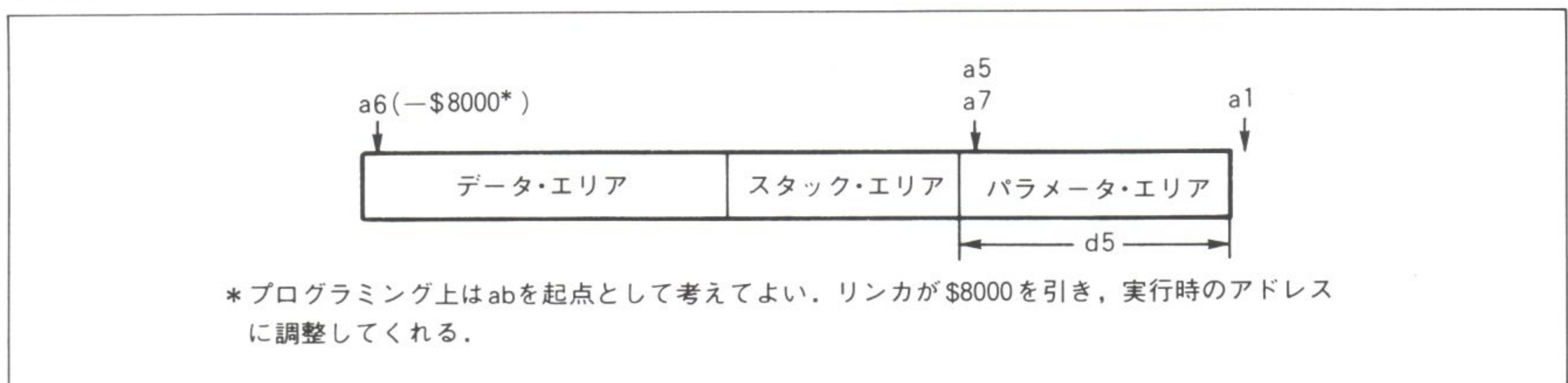
```
move.l    data(a6),d0     (vsect から)
move.w    max(pcr),dl     (psect から)
```

のように書きます．pcr は PC 相対形式を表わし，アセンブラがその命令との相対位置を計算してディスプレースメントを決めてくれます．このように，OS-9の要求するメモリ・データの参照の仕方は定型化されいて，慣れてしまえば簡単です．

プログラムの基本的な考え方は，絶対アドレスを使ってはいけないということで，ジャンプ命令などは PC 相対形式か，あるいは結果的に PC 相対と同じ動作となるパターンでしか利用することができません．しかし，最も簡単にはブランチ命令を使えば PC 相対動作をしてくれるので，この条件も楽にクリアできます．

●付図６　プログラム起動時のレジスタ値とデータ・エリアの関係

a6(−$8000＊)
a5
a7
a1
データ・エリア
スタック・エリア
パラメータ・エリア
d5

＊プログラミング上はa6を起点として考えてよい．リンカが$8000を引き，実行時のアドレスに調整してくれる．

10 システム・コール

OS-9のシステム・コールは，Human68K でスタックを介したパラメータの引き渡しを行なうのに対し，レジスタ・インターフェースによっている点が特徴です．

どちらの方法が良いとかという点については，いろいろ議論のあるところですが，スタックで渡す際も，パラメータを生成するのにレジスタを作業用に使わなければならないこともあり，ユーザ側に戻ってから

スタックを開放する手続きが面倒なことも加味すると，スタック・インターフェースが必ずしも有利とはいえません．レジスタ・インターフェースの欠点は，レジスタの用途を限定するところにありますが，使用レジスタは番号の小さいほうに集中しています．これらのレジスタのデータは，いわば使い捨てにされることが多いので，ユーザ・プログラムに支障をきたす心配はないといえます．このように分析してみると，筆者はむしろレジスタ・インターフェースのほうが使いやすいのではないかと思います．

OS-9のシステム・コールには，**ユーザ・モード・システム・コール，I/O システム・コール，システム・モード・システム・コール**の 3 種類があり，このうちユーザ・モードで使えるのはシステム・コールとI/O システム・コールです．

以上の中で，どのプログラムでも必ずといってよいほど使われるのが，**F$Exit** システム・コールです．このシステム・コールはプログラムを終了させ，OS に制御を戻すもので，正常終了のときはエラー・コードを 0 にします．すなわち

```
clr.w   d1
os9     F$Exit
```

のように記述します．ここで os9擬似命令はシステム・コールを表わします．

ファイルの操作関係では，Human68K のファイル・ハンドルに相当するのが**パス番号**です．ファイルをオープンしたら，クローズまでその番号が有効なので，データ領域などで記憶させておく必要があります．

なお，暗黙に決められているパス番号は次のとおりです．

0……標準入力パス
1……標準出力パス
2……標準エラー出力パス

これらのパスは，ユーザ・プログラムでオープンしたり，クローズしたりする必要がありません．

11　パラメータの省略も可能なマクロ

OS-9アセンブラの**マクロ**は，

```
〈マクロ名〉  MACRO   (マクロ開始)
                ⟆
              ENDM   (マクロ終了)
```

の間に書かれます．

この中でパラメータは n 番目のものを **¥n** と表わし，**¥Ln** がその字数，**¥#**はパラメータの個数としてそれぞれ参照できます．ということは，パラメータの部分省略，全体省略が可能であることを意味します．

●付表 6　アセンブル制御の種類

命　令	記　　述
IFEQ	オペランド＝0 なら真
IFNE	オペランド≠0 なら真
IFLT	オペランド＜0 なら真
IFLE	オペランド≦0 なら真
IFGT	オペランド＞0 なら真
IFGE	オペランド≧0 なら真
IFP1	オペランドが最初のパスなら真(オペランドなし)

●付表7　今回使用するマクロの仕様

型　　　式	機　　　能
Loop	○ループの開始
brklp [〈飛び先〉]	○ループからの脱出 飛び先：脱出時の飛び先ラベル．省略するとRTSでリターンする．
endlp	○ループする処理記述の終了．
open　Read. Write. ⋮ (アクセス・モード) [〈パス記述アドレス〉], [〈パス番号ストア相対アドレス〉], [〈エラー時飛び先〉]	○ファイルのオープン パス記述アドレス：パス記述の位置を示すアドレス記述．省略すると“a5”が採用される．このとき実行後の更新値がa5に入る． パス番号ストア相対アドレス：オープン時に取得したパス番号をストアするアドレスのa6相対値を記述する．省略するとストアされない． エラー時飛び先：オープン時にエラーがあった場合の飛び先．省略すると飛ばない．
io　Read Write ReadLn WritLn 〈パス番号相対アドレス〉, 〈最大バイト数〉, 〈バッファ相対アドレス〉 [, 〈実績バイト数相対アドレス〉] [, 〈エラー時飛び先〉]]	○ファイルの入出力 パス番号相対アドレス：入出力処理に対応するファイル番号のアドレスのa6相対値を記述する． 最大バイト数：入出力する最大バイト数のアドレス記述．直接数値を指定するときは＃nn形式とする． バッファ相対アドレス：入出力バッファのアドレスをa6相対値で記述する． 実績バイト数相対アドレス：入出力した実際のバイト数をストイするアドレスのa6相対値を記述する．省略するとストアされない． エラー時飛び先：入出力エラー（ファイル終了を含む）が発生したときの飛び先．省略すると飛ばない．
close　〈パス番号相対アドレス〉 [, 〈エラー時飛び先〉]	○ファイルのクローズ パス番号相対アドレス：クローズするファイルのパス番号が収容されているアドレスのa6相対値を記述する． エラー時飛び先：エラーが発生した時の飛び先．省略すると飛ばない．
mpsect　〈セクション名〉, 〈スタック・サイズ〉 〈エントリ・ポイント〉	○メインとなるセクションのpsect セクション名：psetのセクション名 スタック・サイズ：psetのスタック・サイズ エントリ・ポイント：psetのエントリ・ポイント ※このマクロはpsetの記述を減らすためのもので，トラップ・エントリは指定できない．
exit　[〈エラー・コード〉]	○プログラムの終了 エラーコード：F$Exitのエラー・コード記述を“＃0”のように指定する．コードが収容されているアドレスを“err(a6)”のように書いてもよく，省略すると直前のd1の値が使われる．

パラメータの内容によって行なうアセンブル制御には，たとえば次のようなものが使えます．

ifeq　¥Ln………パラメータ n が省略されているとき
ifne　¥Ln………パラメータ n が指定されているとき
ifeq　¥#−m…　パラメータが m 個指定されているとき

これらの次に成立時の命令群を並べ，不成立時のものも並記したいときは，

```
else
```

の次に記述します．アセンブル制御の最後は，

```
endc
```

で結びます．アセンブル制御の種類は付表6を参照してください．

以上の機能を利用して，付表7に示す仕様のマクロを実際に作成した結果が付図7です．

この中でioマクロは，システム・コールのI$Read，I$Write，I$Readln，I$Writlnのそれぞれが同様なレジスタ・インターフェースの定義になっていることを利用して1つにまとめたものです．システム・コールは

```
os9　I$xxxx
```

の形式で行ないますが，xxxxの部分をパラメータで与えるため，マクロでは

●付図７　付表９の仕様に従って作成されたマクロ

```
 opt -l
loop MACRO
 jsr 2(pc)
 addi.l #-4,(sp)
 ENDM

brklp MACRO
 lea 4(sp),sp
 ifeq ¥#
 rts
 else
 bra.s ¥1
 endc
 ENDM

endlp MACRO
 rts
 ENDM

open MACRO
 move.b #¥1,d0
 ifeq ¥L2
 move.l a5,a0
 else
 move.l ¥2(a6),a0
 endc
 os9 I$Open
 ifeq ¥#-4
 bcs.s ¥4
 endc
 ifeq ¥L2
 move.l a0,a5
 endc
 ifne ¥L3
 move.w d0,¥3(a6)
 endc
 ENDM

io MACRO
 move.w ¥2(a6),d0
 move.l ¥3,d1
 lea ¥4(a6),a0
 os9 I$¥1
 ifeq ¥#-6
 bcs.s ¥6
 endc
 ifgt ¥#-4
 move.l d1,¥5(a6)
 endc
 ENDM

close MACRO
 move.w ¥1(a6),d0
 os9 I$Close
 ifeq ¥#-2
 bcc.s ¥2
 endc
 ENDM

mpsect MACRO
Typ_Lang set (Prgrm<<8)+Objct
Attr_Rev set (ReEnt<<8)+Revision
 psect ¥1,Typ_Lang,Attr_Rev,Edition,¥2,¥3
 ENDM

exit MACRO
 ifeq ¥#-1
 move.w ¥1,d1
 endc
 os9 F$Exit
 ENDM
 opt l
```

os9　I＄¥1　(パラメータ1を代入)

と記述しています．

loop と endlp はループ(繰り返し)の先頭と末尾に用いるためのもので，brklp はループから脱出するときに使います．このループではスタックを利用しているので，ループ内ループといった階層構造をとることができます．

12 コマンド・パラメータの取り込み

OS-9のプログラムが起動された直後のレジスタ値については，すでに説明したとおりですが，このうちA5はパラメータ・エリアの先頭，D5はパラメータ・サイズを指しています．したがって，この位置から後方にコマンド・パラメータが並んでいるので，コマンド・パラメータを取得するのはいとも簡単です．

付図8に示すプログラムは，コマンド・パラメータの内容(全文字列)をそのまま標準出力に書き出すものです．OS-9では，echo コマンドが同様な機能をもっています．このプログラムは，いわば私設 echo コマンドのためのものですが，入門者がOS-9のアセンブラ・プログラムとして最初に取り組むサンプルには適当でしょう．

プログラムは，最初にI＄writln の各パラメータをセットするために

　Sout(標準出力のパス番号＝1)　……D0
　A5(文字列アドレス)　…………………A0
　D5(文字数)　……………………………D1

の転送を行ないます．I＄writln 実行後，C フラグの値を参照してエラーがなかったかどうかを確認し，正常なら

　Noerr(エラーコード＝0)……………>D1

●付図8　私設 echo コマンド（ec）のプログラム

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  15:12  Page    1
 ec.a
Ec -
00001                          nam      Ec
00002 *
00003 *      Easy Echo command Program
00004 *
00005                          use      ../defs/defsfile
00001                          opt      -l
00006
00007  00000001 Edition        equ      1
00008  00000101 Typ_Lang       equ      (Prgrm<<8)+Objct
00009  00008001 Attr_Rev       equ      (ReEnt<<8)+1
00010                          psect    Ec,Typ_Lang,Attr_Rev,Edition,256,Echo
00011
00012  00000001 Sout           equ      1
00013  00000000 Noerr          equ      0
00014
00015 0000 303c Echo           move.w   #Sout,d0       Standard-output
00016 0004 204d                move.l   a5,a0          Parameter-address
00017 0006 2205                move.l   d5,d1          Parameter-size
00018 0008=4e40                os9      I$WritLn       Display
00019 000c 6504                bcs.s    exit           If eroor
00020 000e 323c                move.w   #Noerr,d1      No-error
00021 0012=4e40 exit           os9      F$Exit         End of program
00022
00023  00000016                ends
00024
```

の転送を行ないます．エラーのあるときはD1にエラー・コードが与えられているので，そのままF$exitを実行して終了します．

13 文字列の入出力

OS-9では，ファイルの内容を表示するコマンドはtype(Human68K)ではなく，listです．このコマンドの処理は，ファイルから文字列を読み出して出力することの繰り返しなので非常に簡単です．ここでは私設listコマンドのプログラムを紹介し，文字列データの入出力のサンプルをお目にかけます．

付図9はこのために作成したプログラムmylistで，先述のマクロを使って組んだため，少ないステップでできています．プログラムは，入力ファイルをオープンし，ループ・マクロで1行ずつの入力，出力を繰り返しています．そして，入力ファイル終了時にはクローズして，プログラム終了となります．

ここではマクロを使っているためこんなに簡単になっていますが，このままではシステム・コール時のレジスタへのパラメータの引き渡しの状況がわからないので，r68(アセンブラ)の_xオプション(マクロ展開部分も出力する)を指定したものを付図10に示します．

リスト中の＋表示のある部分がマクロ展開部分です．

文字列の入出力にはI$Readln(入力)，I$Writlnが使われ，これらのシステム・コールは最大文字数が$0dを検出すると終了します．実際に入出力した字数は，D1にロング・ワード・サイズで与えられます．

●付図9　私設listコマンド（mylist）のプログラム

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  15:12  Page     1
 mylist.a
mylist -
00001                           nam      mylist
00002 *
00003 *      ﾏｸﾛ ｦ ﾂｶｯﾃ ﾘｽﾄ ｦ ｼｭﾂﾘｮｸ ｽﾙ
00004 *
00005                           use      ../defs/defsfile
00001                           opt      -l
00006                           use      macro
00001                           opt      -l
00007  00000001 Revision        equ      1
00008  00000001 Edition         equ      1
00009  00000000 normal          equ      0
00010
00011                           mpsect   mact,512,start
00012                           vsect
00013  00000000 inpath          ds.w     1
00014 0000 0001 outpath         dc.w     1
00015  00000002 length          ds.l     1
00016  00000006 buf             ds.b     256
00017  00000002                 ends
00018
00019 0000 610c start           bsr.s    prelist
00020 0002 611c                 bsr.s    list_main
00021 0004 6154                 bsr.s    postlist
00022                           exit     #normal
00023
00024           prelist         open     Read_,,inpath
00025 001e 4e75                 rts
00026
00027           list_main       loop
00028                           io       ReadLn,inpath,#256,buf,length,atEof
00029                           io       WritLn,outpath,length(a6),buf
00030                           endlp
00031           atEof           brklp
00032
00033           postlist        close    inpath
00034 0062 4e75                 rts
00035
00036  00000064                 ends
00037
```

入力のみパス番号割り当てをシステムに依存するファイルを使用しているので，オープンしたとき D0にワード・サイズで与えられた値を inpath に記憶し，以下の I$ Readln，I$ Close に使っています．

●付図10　付図 9 のプログラムをマクロ展開したもの

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  16:48  Page      1
 mylist.a
mylist -
00001                         nam      mylist
00002 *
00003 *       ﾏｸﾛ ｦ ﾂｶｯﾃ ﾘｽﾄ ｦ ｼｭﾂﾘｮｸ ｽﾙ
00004 *
00005                         use      ../defs/defsfile
00001                         opt      -l
00006                         use      macro
00001                         opt      -l
00007  00000001 Revision      equ      1
00008  00000001 Edition       equ      1
00009  00000000 normal        equ      0
00010
00011                         mpsect   mact,512,start
00012  00000101+Typ_Lang      set      (Prgrm<<8)+Objct
00013  00008001+Attr_Rev      set      (ReEnt<<8)+Revision
00014          +              psect    mact,Typ_Lang,Attr_Rev,Edition,512,start
00015                         vsect
00016  00000000 inpath        ds.w     1
00017 0000 0001 outpath       dc.w     1
00018  00000002 length        ds.l     1
00019  00000006 buf           ds.b     256
00020  00000002               ends
00021
00022 0000 610c start         bsr.s    prelist
00023 0002 611c               bsr.s    list_main
00024 0004 6154               bsr.s    postlist
00025                         exit     #normal
00027 0006 323c+              move.w   #normal,d1
00029 000a=4e40+              os9      F$Exit
00030
00031          prelist        open     Read_,,inpath
00032 000e 103c+              move.b   #Read_,d0
00034 0012 204d+              move.l   a5,a0
00038 0014=4e40+              os9      I$Open
00043 0018 2a48+              move.l   a0,a5
00046 001a 3d40+              move.w   d0,inpath(a6)
00048 001e 4e75               rts
00049
00050          list_main      loop
00051 0020 4eba+              jsr      2(pc)
00052 0024 0697+              addi.l   #-4,(sp)
00053                         io       ReadLn,inpath,#256,buf,length,atEof
00054 002a 302e+              move.w   inpath(a6),d0
00055 002e 223c+              move.l   #256,d1
00056 0034 41ee+              lea      buf(a6),a0
00057 0038=4e40+              os9      I$ReadLn
00059 003c 6516+              bcs.s    atEof
00062 003e 2d41+              move.l   d1,length(a6)
00064                         io       WritLn,outpath,length(a6),buf
00065 0042 302e+              move.w   outpath(a6),d0
00066 0046 222e+              move.l   length(a6),d1
00067 004a 41ee+              lea      buf(a6),a0
00068 004e=4e40+              os9      I$WritLn
00075                         endlp
00076 0052 4e75+              rts
00077          atEof          brklp
00078 0054 4fef+              lea      4(sp),sp
00080 0058 4e75+              rts
00084
00085          postlist       close    inpath
00086 005a 302e+              move.w   inpath(a6),d0

Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  16:48  Page      2
 mylist.a
mylist -
00087 005e=4e40+              os9      I$Close
00091 0062 4e75               rts
00092
00093  00000064               ends
00094
```

14 データ・ブロックの入出力

ディスクの文字列以外のデータ(たとえば実行形式などのプログラム・ファイル)は，文字列データのように区切りがないので，ブロック単位に読み書きするしかありません．当然このようなデータに対する入出力命令は，ブロック・データ専用のものを使うことになります．

付図11は私設 copy コマンドの mycopy プログラムで，入力したブロック・データをそのまま出力に転送します．出力ファイルもディスクになっているので，実行開始時には入出力ともにオープンしなければなりません．このリストではコマンド・パラメータで与えられた入出力ファイル名の処理がなされていないように見えますが，展開形(付図12)を見るとその「水面下」のようすが手に取るようにわかります．

つまり，open マクロではファイル名文字列(パス記述)アドレスを A5から貰い受けるようになっており，その値の更新後のアドレスを A5に転送して戻しています．したがって，入力ファイルの次に指定されているものを出力ファイル名としてオープンするように，内部でリレーされているのです．ですから，このプログラムは，

mycopy　<入力ファイル名> <出力ファイル名>

というコマンド形式に立派に対応できているのです．

●付図11　私設 copy（mycopy）コマンドのプログラム

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  15:12  Page    1
 mycopy.a
mycopy -
00001                         nam      mycopy
00002 *
00003 *
00004                         use      ../defs/defsfile
00001                         opt      -l
00005                         use      macro
00001                         opt      -l
00006  00000001 Revision      equ      1
00007  00000001 Edition       equ      1
00008  00000000 normal        equ      0
00009
00010                         mpsect   mact,1200,start
00011                         vsect
00012  00000000 inpath        ds.w     1
00013  00000002 outpath       ds.w     1
00014  00000004 length        ds.l     1
00015  00000008 buf           ds.b     1024
00016  00000000               ends
00017
00018 0000 610c start         bsr.s    precopy
00019 0002 612c               bsr.s    copy_main
00020 0004 6164               bsr.s    postcopy
00021                         exit     #normal
00022
00023           precopy       open     Read_,,inpath
00024                         open     Write_,,outpath
00025 002e 4e75               rts
00026
00027           copy_main     loop
00028                         io       Read,inpath,#1024,buf,length,atEof
00029                         io       Write,outpath,length(a6),buf
00030                         endlp
00031           atEof         brklp
00032
00033           postcopy      close    inpath
00034                         close    outpath
00035 007a 4e75               rts
00036
00037  0000007c               ends
00038
```

●付図12　付図11のプログラムをマクロ展開したもの

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  16:48  Page    1
 mycopy.a
mycopy -
00001                        nam      mycopy
00002 *
00003 *
00004                        use      ../defs/defsfile
00001                        opt      -l
00005                        use      macro
00001                        opt      -l
00006  00000001 Revision     equ      1
00007  00000001 Edition      equ      1
00008  00000000 normal       equ      0
00009
00010                        mpsect   mact,1200,start
00011  00000101+Typ_Lang     set      (Prgrm<<8)+Objct
00012  00008001+Attr_Rev     set      (ReEnt<<8)+Revision
00013         +              psect    mact,Typ_Lang,Attr_Rev,Edition,1200,start
00014                        vsect
00015  00000000 inpath       ds.w     1
00016  00000002 outpath      ds.w     1
00017  00000004 length       ds.l     1
00018  00000008 buf          ds.b     1024
00019  00000000              ends
00020
00021 0000 610c start        bsr.s    precopy
00022 0002 612c              bsr.s    copy_main
00023 0004 6164              bsr.s    postcopy
00024                        exit     #normal
00026 0006 323c+             move.w   #normal,d1
00028 000a=4e40+             os9      F$Exit
00029
00030           precopy      open     Read_,,inpath
00031 000e 103c+             move.b   #Read_,d0
00033 0012 204d+             move.l   a5,a0
00037 0014=4e40+             os9      I$Open
00042 0018 2a48+             move.l   a0,a5
00045 001a 3d40+             move.w   d0,inpath(a6)
00047                        open     Write_,,outpath
00048 001e 103c+             move.b   #Write_,d0
00050 0022 204d+             move.l   a5,a0
00054 0024=4e40+             os9      I$Open
00059 0028 2a48+             move.l   a0,a5
00062 002a 3d40+             move.w   d0,outpath(a6)
00064 002e 4e75              rts
00065
00066           copy_main    loop
00067 0030 4eba+             jsr      2(pc)
00068 0034 0697+             addi.l   #-4,(sp)
00069                        io       Read,inpath,#1024,buf,length,atEof
00070 003a 302e+             move.w   inpath(a6),d0
00071 003e 223c+             move.l   #1024,d1
00072 0044 41ee+             lea      buf(a6),a0
00073 0048=4e40+             os9      I$Read
00075 004c 6516+             bcs.s    atEof
00078 004e 2d41+             move.l   d1,length(a6)
00080                        io       Write,outpath,length(a6),buf
00081 0052 302e+             move.w   outpath(a6),d0
00082 0056 222e+             move.l   length(a6),d1
00083 005a 41ee+             lea      buf(a6),a0
00084 005e=4e40+             os9      I$Write
00091                        endlp
00092 0062 4e75+             rts
00093           atEof        brklp
00094 0064 4fef+             lea      4(sp),sp
00096 0068 4e75+             rts
00100
00101           postcopy     close    inpath
00102 006a 302e+             move.w   inpath(a6),d0
00103 006e=4e40+             os9      I$Close
00107                        close    outpath
00108 0072 302e+             move.w   outpath(a6),d0
00109 0076=4e40+             os9      I$Close
00113 007a 4e75              rts
00114
00115  0000007c              ends
00116
```

ブロック・データに対するI/Oシステム・コールは，I$Read(入力)，I$Write(出力)で，これらとクローズのために，オープン時にはinpath，outpathにパス番号を記憶させせます．

なお，オープン，クローズ用のシステム・コールは，ブロック・データだからといって変わることはありません．

15 エラー・メッセージを出力するプログラム

OS-9のシステムから出されるエラー・メッセージは，/d0/sys/errmsgに収容されています．この内容はF$PErrで出力できるので，自作のプログラム中でエラー表示に使用すると便利です．

付図13は，キーボードから入力した3桁のエラー番号(10進値)を2進値に変換して，その値でF$PErrを働かせるものです．このシステム・コールのエラー・コードはD1で受けるようになっているので，他のシステム・コールでエラーが検出されたら，そのままD1の値を引き継げばよいようになっています．

エラー・メッセージ・ファイルは上記のものに限らず，同じフォーマットならば任意のものが使えます．このプログラムのように，そのファイルをオープンしておいて，そのパス番号をD0に入れてエラー番号(D1)とともにF$PErrに送り込めばよいのです．

このプログラムでは，キーボードから入力したエラー番号が10進数以外のものだったときは内部からエラー・メッセージを出して正常終了するようになっています．エラー番号の入力がコマンド・パラメータなのでこうせざるを得ないのですが，他のプログラムでも，F$PErrシステム・コールを使うのはコマンド・パラメータに起因するケースが多いと考えられます．

●付図13　エラー・メッセージを出力するプログラム

```
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  16:48  Page      1
 printerr.a
printerr -
00001                          nam      printerr
00002                          use      ../defs/oskdefs.d
00001                          opt      -l
00003                          use      macro
00001                          opt      -l
00004
00005  00000001 Revision       equ      1
00006  00000001 Edition        equ      1
00007  00000000 normal         equ      0
00008
00009                          mpsect   printerr,512,entry
00010  00000101+Typ_Lang       set      (Prgrm<<8)+Objct
00011  00008001+Attr_Rev       set      (ReEnt<<8)+Revision
00012         +                psect    printerr,Typ_Lang,Attr_Rev,Edition,512,entry

00013                          vsect
00014  00000000 ipath          ds.w     1
00015 0000 0002 errpath        dc.w     2
00016 0002 2f64 fileid         dc.b     "/d0/sys/errmsg",0
00017 0011 4261 errmsg         dc.b     "Bad Error-number. Try again!",$0d
00018  0000002e                ends
00019
00020 *
00021 * Main part
00022 *
00023 0000 4241 entry          clr.w    d1
00024 0002 343c                move.w   #1,d2
00025 0006 224d                movea.l  a5,a1
00026 0008 3a3c                move.w   #3,d5
00027 000c 5689                addq.l   #3,a1
00028 000e 612c                bsr.s    dtb
00029 0010 4bee                lea      fileid(a6),a5
00030                          open     Read_,,ipath
00031 0014 103c+               move.b   #Read_,d0
00033 0018 204d+               move.l   a5,a0
00037 001a=4e40+               os9      I$Open
00042 001e 2a48+               move.l   a0,a5
```

```
00045 0020 3d40+            move.w   d0,ipath(a6)
00047 0024 302e             move.w   ipath(a6),d0
00048 0028=4e40             os9      F$PErr
00049                       close    ipath
00050 002c 302e+            move.w   ipath(a6),d0
00051 0030=4e40+            os9      I$Close
00055                       exit     #normal
00057 0034 323c+            move.w   #normal,d1
00059 0038=4e40+            os9      F$Exit
Microware OS-9/68000 Resident Macro Assembler V1.6  88/09/29  16:48  Page     2
 printerr.a
printerr -
00061 *
00062 * Decimal to Binary
00063 *
00064          dtb        loop
00065 003c 4eba+            jsr      2(pc)
00066 0040 0697+            addi.l   #-4,(sp)
00067 0046 1021             move.b   -(a1),d0
00068 0048 0c00             cmpi.b   #'0',d0
00069 004c 6520             blo.s    err
00070 004e 0c00             cmpi.b   #'9',d0
00071 0052 621a             bhi.s    err
00072 0054 903c             sub.b    #$30,d0
00073 0058 c0c2             mulu.w   d2,d0
00074 005a d240             add.w    d0,d1
00075 005c c4fc             mulu.w   #10,d2
00076 0060 9a7c             sub.w    #1,d5
00077 0064 6702             beq.s    return
00078                       endlp
00079 0066 4e75+            rts
00080          return     brklp
00081 0068 4fef+            lea      4(sp),sp
00083 006c 4e75+            rts
00087 *
00088 * Error parameter handler
00089 *
00090          err        io       WritLn,errpath,#29,errmsg
00091 006e 302e+            move.w   errpath(a6),d0
00092 0072 223c+            move.l   #29,d1
00093 0078 41ee+            lea      errmsg(a6),a0
00094 007c=4e40+            os9      I$WritLn
00101                       exit     #normal
00103 0080 323c+            move.w   #normal,d1
00105 0084=4e40+            os9      F$Exit
00106
```

おわりに

BASIC から C，アセンブラと，X68000では各言語が有機的に連動しています．

本書ではこのことに着目し，BASIC の外部関数をアセンブラで作成したり，言語間の変換や他言語プログラムの挿入などについて，実例をあげて紹介してきました．

最近のプログラミングの傾向としては，C を中心に使い，面倒なところはアセンブラでカバーするといった手法がよく使われているようです．いわばアセンブラは「奥の手」であって，古い言語とはいえまだまだ引退することは考えられません．

X68000のプログラミングの終着駅がアセンブラであることは明らかですが，筆者に言わせると68000CPU のアセンブラの終着駅は，やはり OS-9のそれということになります．

第5章ではスペースの都合もあって，アセンブラ・プログラムのサンプルをたくさん掲載することができなかったのですが，その代わり巻末の「OS-9/68000への招待」のところで OS-9流のアセンブラ・プログラムを紹介しています．

本格的に68000アセンブラ・プログラムを楽しみたい読者は，是非 OS-9に挑戦してみてください．

参考文献

(1) シャープ「X68000技術資料」1986年11月20日

(2) モトローラ「M68000 8-/16-/32-Bit Microprocessors Programmer's Reference Manual」1986年第4版

(3) モトローラ「MC68000 16Bit Microprocessors User's Manual」第4版　CQ 出版社

(4) シャープ「X68000取扱説明書」

(5) 千葉憲昭「FM シリーズいざという時の事典」1987年初版　ナツメ社

(6) シャープ「Human68K ユーザーズマニュアル」

(7) シャープ「X68000 BASIC マニュアル」

(8) シャープ「PRO-68K C リファレンスマニュアル」

(9) シャープ「PRO-68K C ライブラリマニュアル」

(10) シャープ「PRO-68K アセンブラマニュアル」

(11) 千葉憲昭「68000システムの製作全科(上)」昭和63年9月初版　技術評論社

(12) シャープ「PRO-68K プログラマーズマニュアル」

(13) 千葉憲昭「OS-9/68000のすべて」The BASIC 1987年6～12月号（連載）　技術評論社

(14) 日本電気「ユーザーズ・マニュアル μPD72065，72065,72066 CMOS FDC　IEM-915B（第3版）January 1989 P

索　引

英数字和文項目

関連コマンド

千葉憲昭（ちば　のりあき）

1946年　北海道に生まれる
1969年　福岡工大電子工学科卒
同　年　北海道庁電子計算課勤務
1984年　著述家に転向，現在に至る．
(財)札幌エレクトロニクス・センター運営委員
著　書　『マイコンピュータNo.6，8，17，21』（共著，CQ出版），『6809マシン語スタディ』（CQ出版），『BASICユーティリティ・プログラミング』（CQ出版），『FMシリーズいざ！という時の事典』（ナツメ社），『68000システムの製作全科（上，下）』（技術評論社）他多数

X68000ベスト・プログラミング入門

平成元年3月5日　初版　第1刷発行

著　者　千葉憲昭
発行者　片岡　巌
発行所　株式会社技術評論社
東京都千代田区九段南2-4-13
電話　03（262）9351　営業部
03（262）7671　編集部
印刷／製本　株式会社 金羊社

定価はカバーに表示してあります

ISBN4-87408-992-5　C3055

X68000

X68000ベスト・プログラミング入門

千葉憲昭 著

技術評論社

X68000

X68000ベスト・プログラミング入門

千葉憲昭 著

技術評論社

ISBN4-87408-992-5 C3055 ¥2800E 定価 2800円